Tell us about your PDF experience.

C/C++ projects and build systems in Visual Studio

Article • 12/09/2021

You can use Visual Studio to edit, compile, and build any C++ code base with full IntelliSense support without having to convert that code into a Visual Studio project or compile with the MSVC toolset. For example, you can edit a cross-platform CMake project in Visual Studio on a Windows machine, then compile it for Linux using g++ on a remote Linux machine.

C++ compilation

To *build* a C++ program means to compile source code from one or more files and then link those files into an executable file (.exe), a dynamic-load library (.dll) or a static library (.lib).

Basic C++ compilation involves three main steps:

- The C++ preprocessor transforms all the #directives and macro definitions in each source file. This creates a *translation unit*.
- The C++ compiler compiles each translation unit into object files (.obj), applying whatever compiler options have been set.
- The *linker* merges the object files into a single executable, applying the linker options that have been set.

The MSVC toolset

The Microsoft C++ compiler, linker, standard libraries, and related utilities make up the MSVC compiler toolset (also called a toolchain or "build tools"). These are included in Visual Studio. You can also download and use the command-line toolset as a free standalone package. For more information, see Build Tools for Visual Studio on the Visual Studio Downloads page.

You can build simple programs by invoking the MSVC compiler (cl.exe) directly from the command line. The following command accepts a single source code file, and invokes cl.exe to build an executable called *hello.exe*:

Windows Command Prompt

```
cl /EHsc hello.cpp
```

Here the compiler (cl.exe) automatically invokes the C++ preprocessor and the linker to produce the final output file. For more information, see Building on the command line.

Build systems and projects

Most real-world programs use some kind of *build system* to manage complexities of compiling multiple source files for multiple configurations (debug vs. release), multiple platforms (x86, x64, ARM, and so on), custom build steps, and even multiple executables that must be compiled in a certain order. You make settings in a build configuration file(s), and the build system accepts that file as input before it invokes the compiler. The set of source code files and build configuration files needed to build an executable file is called a *project*.

The following list shows various options for Visual Studio Projects - C++:

- create a Visual Studio project by using the Visual Studio IDE and configure it by using property pages. Visual Studio projects produce programs that run on Windows. For an overview, see Compiling and Building in the Visual Studio documentation.
- open a folder that contains a CMakeLists.txt file. CMake support is integrated into Visual Studio. You can use the IDE to edit, test, and debug without modifying the CMake files in any way. This enables you to work in the same CMake project as others who might be using different editors. CMake is the recommended approach for cross-platform development. For more information, see CMake projects.
- open a loose folder of source files with no project file. Visual Studio will use heuristics to build the files. This is an easy way to compile and run small console applications. For more information, see Open Folder projects.
- open a folder that contains a makefile, or any other build system configuration file. You can configure Visual Studio to invoke any arbitrary build commands by adding JSON files to the folder. For more information, see Open Folder projects.
- Open a Windows makefile in Visual Studio. For more information, see NMAKE Reference.

MSBuild from the command line

You can invoke MSBuild from the command line by passing it a .vcxproj file along with command-line options. This approach requires a good understanding of MSBuild, and is

recommended only when necessary. For more information, see MSBuild.

In This Section

Visual Studio projects

How to create, configure, and build C++ projects in Visual Studio using its native build system (MSBuild).

CMake projects

How to code, build, and deploy CMake projects in Visual Studio.

Open Folder projects

How to use Visual Studio to code, build, and deploy C++ projects based on any arbitrary build system, or no build system at all.

Release builds

How to create and troubleshoot optimized release builds for deployment to end users.

Use the MSVC toolset from the command line

Discusses how to use the C/C++ compiler and build tools directly from the command line rather than using the Visual Studio IDE.

Building DLLs in Visual Studio

How to create, debug, and deploy C/C++ DLLs (shared libraries) in Visual Studio.

Walkthrough: Creating and Using a Static Library

How to create a **.lib** binary file.

Building C/C++ Isolated Applications and Side-by-side Assemblies

Describes the deployment model for Windows Desktop applications, based on the idea of isolated applications and side-by-side assemblies.

Configure C++ projects for 64-bit, x64 targets

How to target 64-bit x64 hardware with the MSVC build tools.

Configure C++ projects for ARM processors

How to use the MSVC build tools to target ARM hardware.

Optimizing Your Code

How to optimize your code in various ways including program guided optimizations.

Configuring Programs for Windows XP

How to target Windows XP with the MSVC build tools.

C/C++ Building Reference

Provides links to reference articles about program building in C++, compiler and linker options, and various build tools.

Visual Studio projects - C++

Article • 10/04/2023

A *Visual Studio project* is a collection of code files and assets such as icons, images, and so on, that are built together using the MSBuild build system. MSBuild is the native build system for Visual Studio and is generally the best build system to use for Windows-specific programs. MSBuild is tightly integrated with Visual Studio, but you can also use it from the command line.

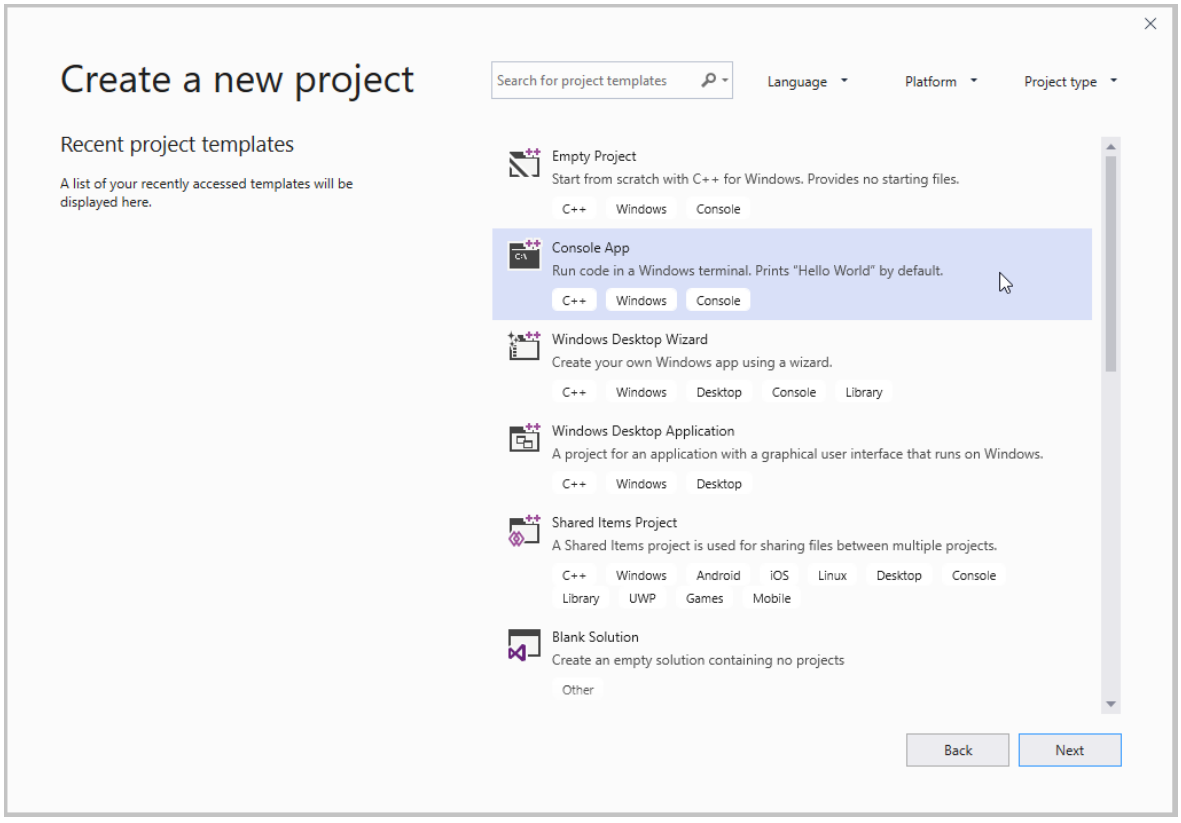
For information about upgrading MSBuild projects from older versions of Visual Studio, see the Microsoft C++ Porting and Upgrading Guide.

For cross-platform projects, or projects that use open-source libraries, we recommend using CMake projects in Visual Studio in Visual Studio 2017 and later.

Create a Visual Studio C++ project

1. Create a C++ project by choosing **File** > **New** > **Project**.

2. In the **Create a new project** dialog, set the **Language** dropdown to **C++**. This filters the list of project templates to C++ projects. You can filter the templates by setting the **Platform**, **Project Type**, or by entering keywords in the search box.



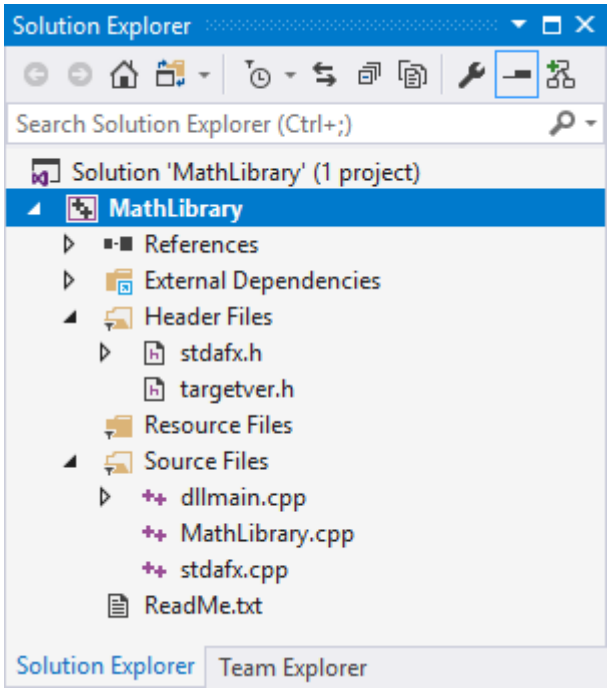
3. Select a project template, then choose **Next**.

4. On the **Configure your new project page**, enter project-specific settings such as the project name or location and then choose **Create** to create your project.

For more information about the default project templates included in Visual Studio, see C++ project templates in Visual Studio.

You can create your own project templates. For more information, see How to: Create project templates.

After you create a project, it appears in the Solution Explorer window:



When you create a new project, a solution file (.sln) is also created. A *Visual Studio solution* is a collection of one or more projects. You can add another project to the solution by right-clicking the solution name in **Solution Explorer** > **Add** > **New project**.

The solution file coordinates build dependencies when you have multiple related projects. Compiler options are set at the project level.

Add code, icons, and other assets to a project

Add source code files, icons, or any other items to your project by right-clicking on the project in **Solution Explorer** and choosing **Add** > **New** or **Add** > **Existing**.

Add third-party libraries to a project

Over 900 C++ open source libraries are available via the vcpkg package manager. Run the Visual Studio integration step to set up the paths to that library when you reference

it from any Visual Studio project.

They're also commercial third-party libraries that you can install. Follow their installation instructions.

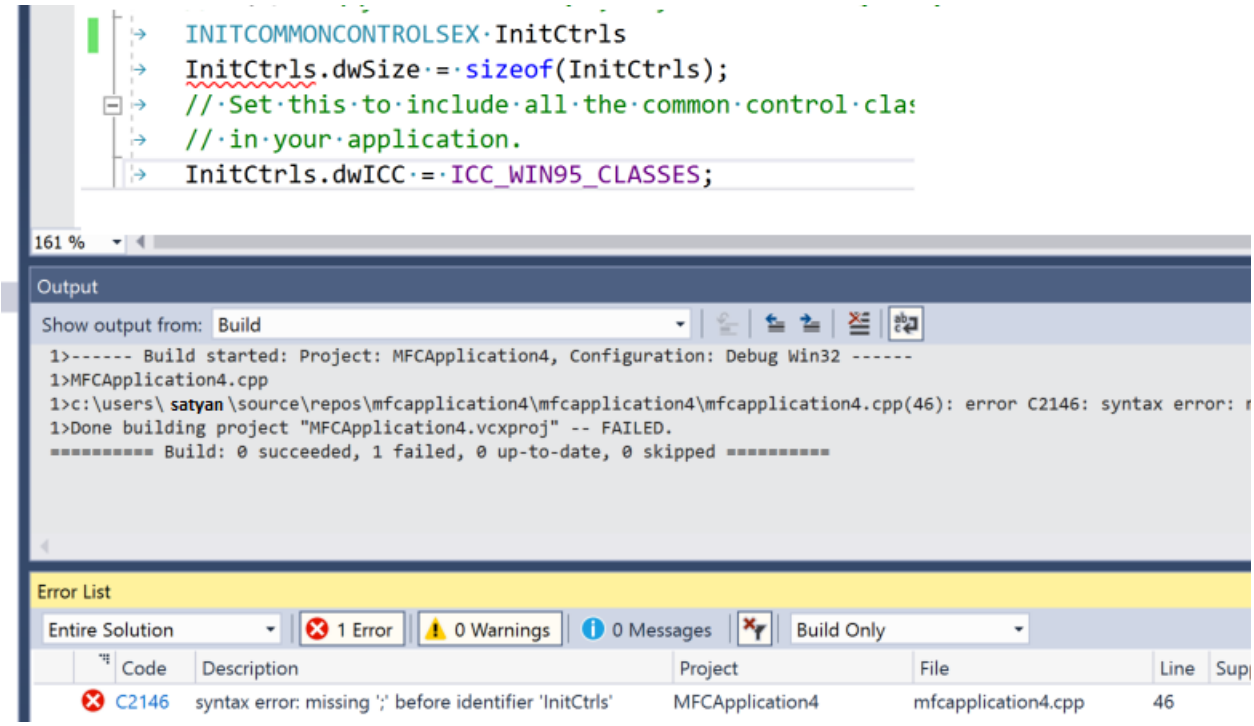
Set compiler options and build properties

To configure build settings for a project, right-click on the project in **Solution Explorer** and choose **Properties**. For more information, see Set C++ compiler and build properties in Visual Studio.

Compile and run a project

To compile and run the new project, press **F5** or click the *debug dropdown* with the green arrow on the main toolbar. The *configuration dropdown* is where you choose whether to perform a *Debug* or *Release* build (or some other custom configuration).

A new project compiles without errors. When adding your own code, you might occasionally introduce an error or trigger a warning. An error prevents the build from completing; a warning doesn't. All errors and warnings appear both in the Output Window and in the Error List when you build the project.



In the **Error List**, you can press **F1** on the highlighted error to go to its documentation topic.

See also

Create a project from existing code
Set C++ compiler and build properties in Visual Studio
Custom build steps and build events
Reference libraries and components at build time
Organize project output files
Projects and build systems
Microsoft C++ porting and upgrade guide

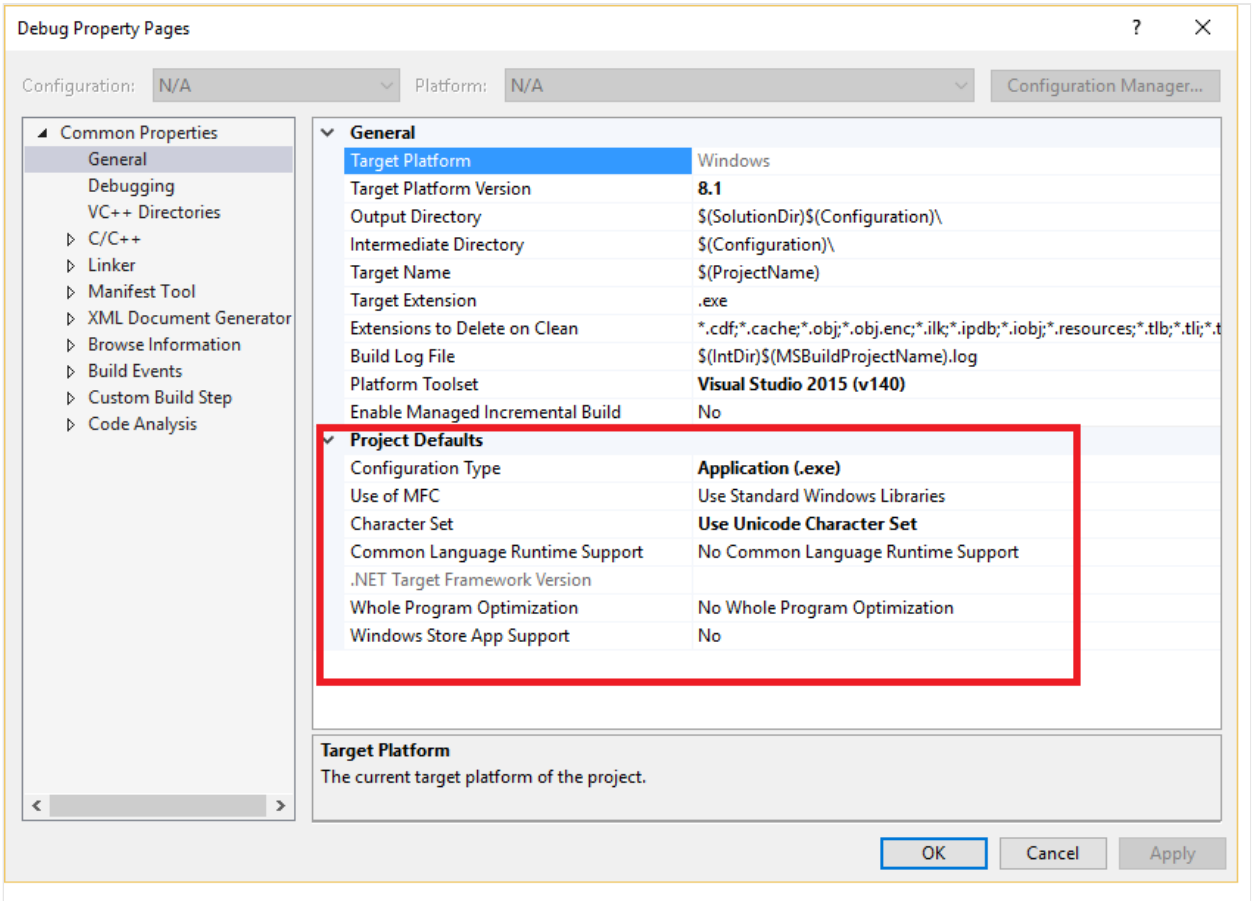
Set compiler and build properties

Article • 03/20/2024

In the IDE, properties expose the information needed to build a project. This information includes the application name, extension (such as DLL, LIB, EXE), compiler options, linker options, debugger settings, custom build steps, and many other things. Typically, you use *property pages* to view and modify these properties. To access the property pages, choose **Project** > ***project-name*** **Properties** from the main menu, or right-click on the project node in **Solution Explorer** and choose **Properties**.

Default properties

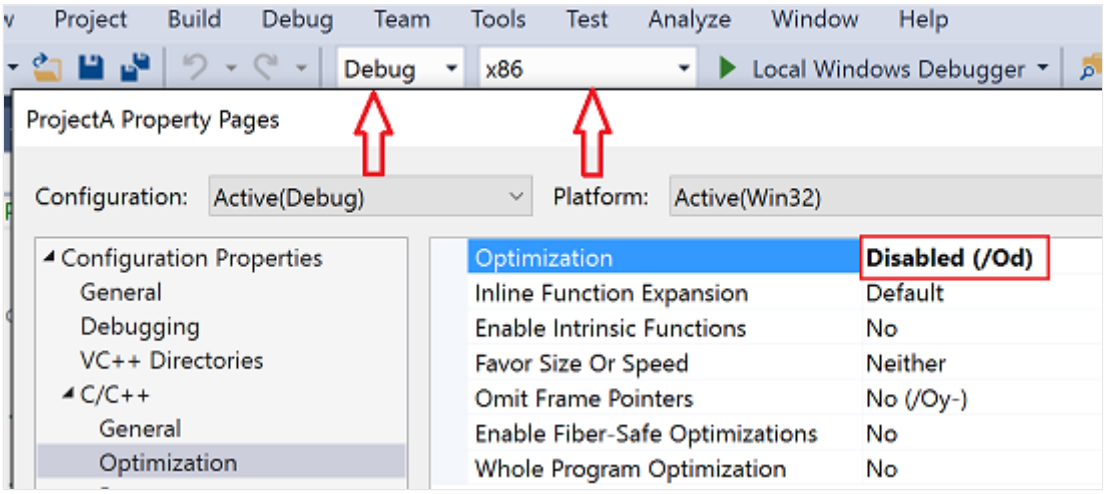
When you create a project, the system assigns values for various properties. The defaults vary somewhat depending on the kind of project and what options you choose in the app wizard. For example, an ATL project has properties related to MIDL files, but these properties are absent in a basic console application. The default properties are shown in the General pane in the Property Pages:



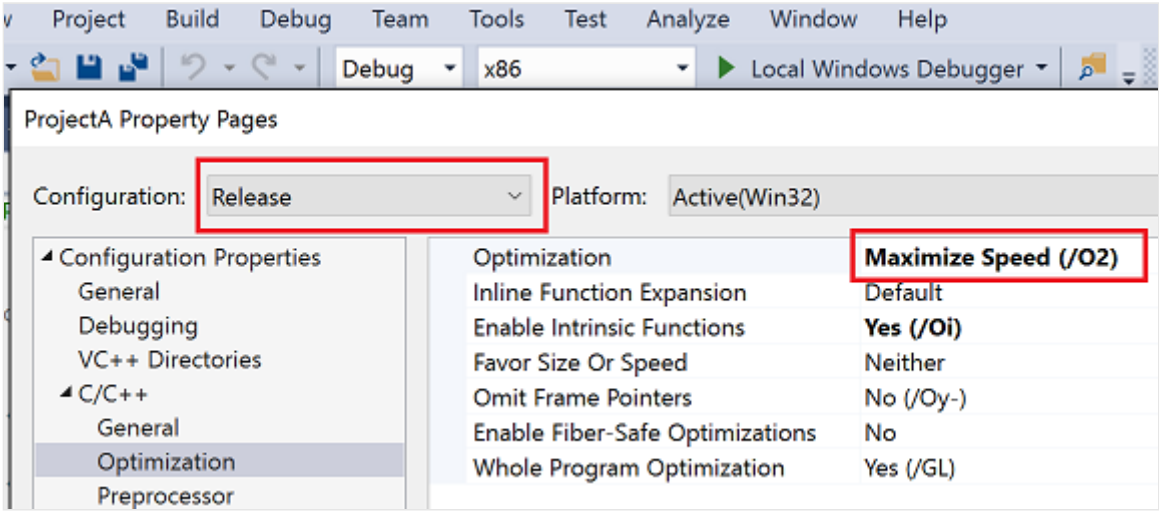
Applying properties to build configurations and target platforms

Some properties, such as the application name, apply to all build variations and target platforms, whether it's a debug or release build. But most properties are configuration-dependent. To generate the correct code, the compiler has to know both the specific platform the program runs on and which specific compiler options to use. So when you set a property, it's important to pay attention to which configuration and platform the new value should apply to. Should it apply only to Debug Win32 builds, or should it also apply to Debug ARM64 and Debug x64? For example, the **Optimization** property, by default, is set to **Maximize Speed (/O2)** in a Release configuration, but is disabled in the Debug configuration.

You can always see and change the configuration and platform a property value should apply to. The following illustration shows the property pages with the configuration and platform information controls at the top. When the **Optimization** property is set here, it only applies to Debug Win32 builds, the currently active configuration, as shown by the red arrows.



The following illustration shows the same project property page, but the configuration has been changed to Release. Note the different value for the Optimization property. Also note that the active configuration is still Debug. You can set properties for any configuration here; it doesn't have to be the active one.



Target platforms

Target platform refers to the kind of device and operating system that the executable will run on. You can build a project for more than one platform. The available target platforms for C++ projects depend on the kind of project. They include but aren't limited to Win32, x64, ARM, ARM64, Android, and iOS. The **x86** target platform that you might see in **Configuration Manager** is identical to **Win32** in native C++ projects. Win32 means 32-bit Windows and **x64** means 64-bit Windows. For more information about these two platforms, see Running 32-bit applications.

The **Any CPU** target platform value that you might see in **Configuration Manager** has no effect on native C++ projects. It's only relevant for C++/CLI and other .NET project types. For more information, see /CLRIMAGETYPE (Specify Type of CLR Image).

For more information about setting properties for a Debug build, see:

- Project settings for a C++ debug configuration
- Debugger Settings and Preparation
- Debugging Preparation: Visual C++ Project Types
- Specify symbol (.pdb) and source files in the Visual Studio debugger

C++ compiler and linker options

C++ compiler and linker options are located under the **C/C++** and **Linker** nodes in the left pane under **Configuration Properties**. These options translate directly to command-line options that are passed to the compiler. To read documentation about a specific option, select the option in the center pane and press **F1**. Or, you can browse documentation for all the options at MSVC compiler options and MSVC linker options.

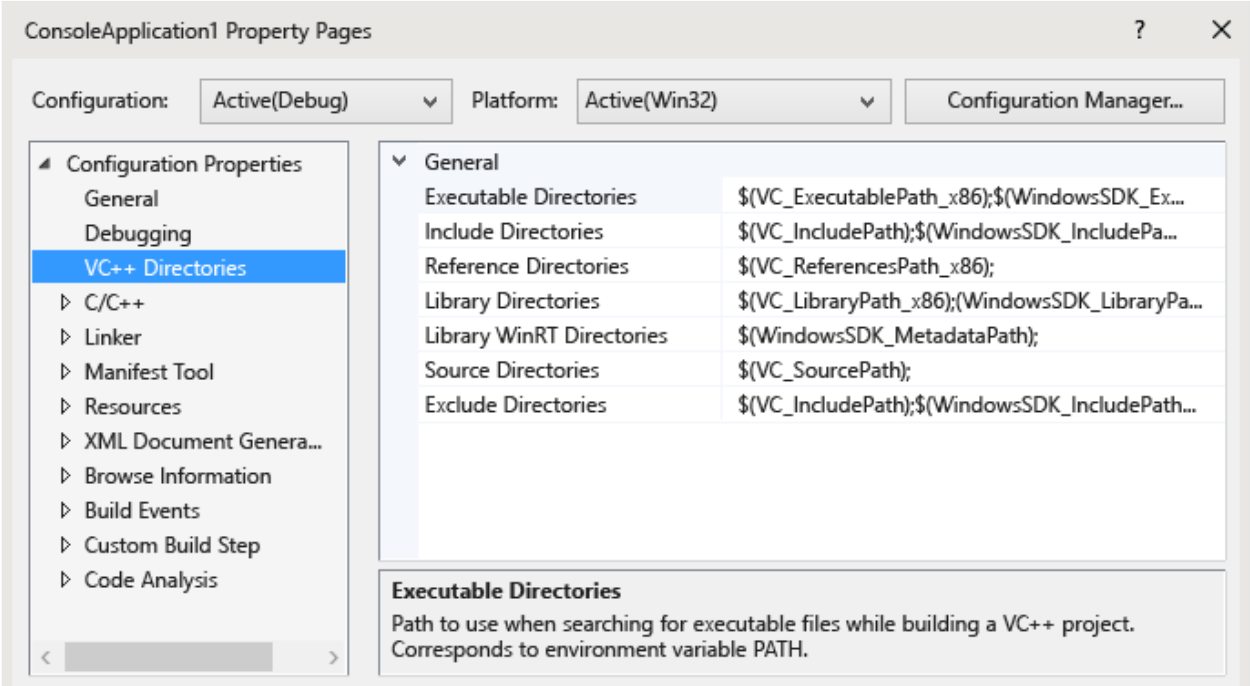
The **Property Pages** dialog box shows only the property pages that are relevant to the current project. For example, if the project doesn't have an `.idl` file, the MIDL property page isn't displayed. For more information about the settings on each property page, see Property Pages (C++).

Directory and path values

MSBuild supports the use of compile-time constants for certain string values, such as include directories and paths, called *macros*. A macro can refer to a value that's defined by Visual Studio or the MSBuild system, or to a user-defined value. Macros look like `$(macro-name)` or `%(item-macro-name)`. They're exposed in the property pages, where you can refer to and modify them by using the Property Editor. Use macros instead of

hard-coded values such as directory paths. Macros make it easier to share property settings between machines and between versions of Visual Studio. And, you can better ensure that your project settings participate correctly in property inheritance.

The following illustration shows the property pages for a Visual Studio C++ project. In the left pane, the **VC++ Directories** *rule* is selected, and the right pane lists the properties that are associated with that rule. The property values are often macros, such as `$(VC_SourcePath)`:



You can use the Property Editor to view the values of all available macros.

Predefined macros

- **Global macros**:
 Global macros apply to all items in a project configuration. A global macro has the syntax `$(name)`. An example of a global macro is `$(VCInstallDir)`, which stores the root directory of your Visual Studio installation. A global macro corresponds to a `PropertyGroup` in MSBuild.

- **Item macros**
 Item macros have the syntax `%(name)`. For a file, an item macro applies only to that file—for example, you can use `%(AdditionalIncludeDirectories)` to specify include directories that apply only to a particular file. This kind of item macro corresponds to an `ItemGroup` metadata in MSBuild. When used in the context of a project configuration, an item macro applies to all files of a certain type. For example, the C/C++ **Preprocessor Definitions** configuration property can take a `%(PreprocessorDefinitions)` item macro that applies to all .cpp files in the project.

This kind of item macro corresponds to an `ItemDefinitionGroup` metadata in MSBuild. For more information, see Item Definitions.

User-defined macros

You can create *user-defined macros* to use as variables in project builds. For example, you could create a user-defined macro that provides a value to a custom build step or a custom build tool. A user-defined macro is a name/value pair. In a project file, use the `$(name)` notation to access the value.

A user-defined macro is stored in a property sheet. If your project doesn't already contain a property sheet, you can create one by following the steps under Share or reuse Visual Studio project settings.

To create a user-defined macro

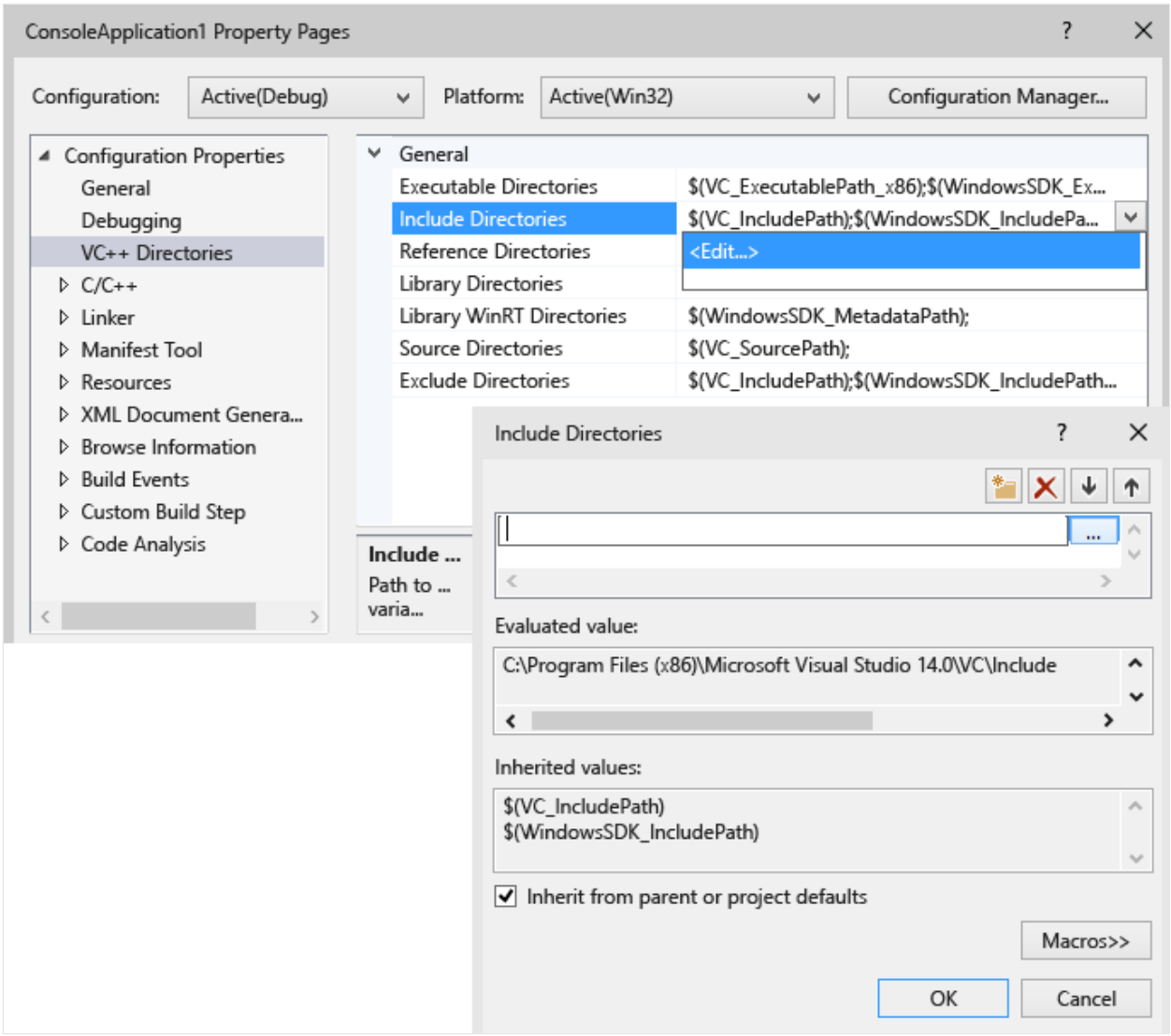
1. Open the **Property Manager** window. (On the menu bar, choose **View** > **Property Manager** or **View** > **Other Windows** > **Property Manager**.) Open the shortcut menu for a property sheet (its name ends in `.user`) and then choose **Properties**. The **Property Pages** dialog box for that property sheet opens.

2. In the left pane of the dialog box, select **User Macros**. In the right pane, choose the **Add Macro** button to open the **Add User Macro** dialog box.

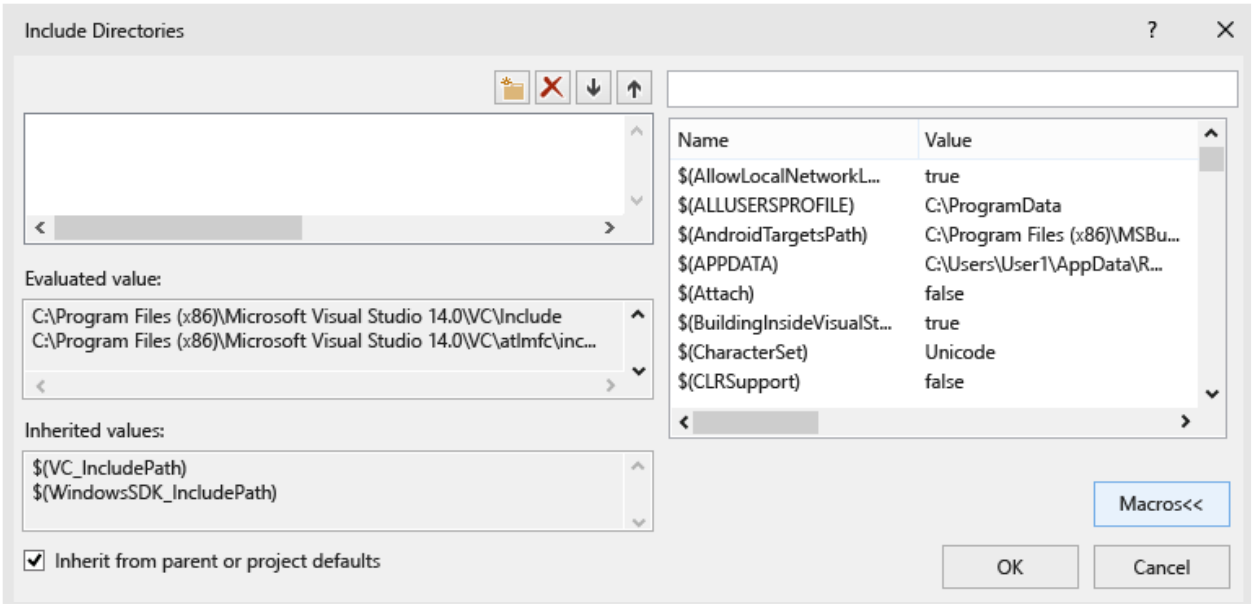
3. In the dialog box, specify a name and value for the macro. Optionally, select the **Set this macro as an environment variable in the build environment** check box.

Property Editor

You can use the Property Editor to modify certain string properties and select macros as values. To access the Property Editor, select a property on a property page and then choose the down arrow button on the right. If the drop-down list contains **<Edit>**, then you can choose it to display the Property Editor for that property.



In the Property Editor, you can choose the **Macros** button to view the available macros and their current values. The following illustration shows the Property Editor for the **Additional Include Directories** property after the **Macros** button was chosen. When the **Inherit from parent or project defaults** check box is selected and you add a new value, it's appended to any values that are currently being inherited. If you clear the check box, your new value replaces the inherited values. In most cases, leave the check box selected.



Add an include directory to the set of default directories

When you add an include directory to a project, it's important not to override all the default directories. The correct way to add a directory is to append the new path, for example "`C:\MyNewIncludeDir\`", and then to Append the **`$(IncludePath)`** macro to the property value.

Quickly browse and search all properties

The **All Options** property page (under the **Configuration Properties** > **C/C++** node in the **Property Pages** dialog box) provides a quick way to browse and search the properties that are available in the current context. It has a special search box and a simple syntax to help you filter results:

No prefix:
Search in property names only (case-insensitive substring).

'`/`' or '`-`':
Search only in compiler switches (case-insensitive prefix)

`v`:
Search only in values (case-insensitive substring).

Set environment variables for a build

The MSVC compiler (cl.exe) recognizes certain environment variables, specifically `LIB`, `LIBPATH`, `PATH`, and `INCLUDE`. When you build with the IDE, the properties that are set in the VC++ Directories Property Page are used to set those environment variables. If `LIB`, `LIBPATH`, and `INCLUDE` values have already been set, for example by a Developer Command Prompt, they're replaced with the values of the corresponding MSBuild properties. The build then prepends the value of the VC++ Directories executable directories property to `PATH`. You can set a user-defined environment variable by creating a user-defined macro and then checking the box that says **Set this macro as an environment variable in the build environment**.

Set environment variables for a debugging session

In the left pane of the project's **Property Pages** dialog box, expand **Configuration Properties** and then select **Debugging**.

In the right pane, modify the **Environment** or **Merge Environment** project settings and then choose the **OK** button.

In this section

Share or reuse Visual Studio project settings
How to create a `.props` file with custom build settings that can be shared or reused.

Project property inheritance
Describes the order of evaluation for the `.props`, `.targets`, `.vcxproj` files, and environment variables in the build process.

Modify properties and targets without changing the project file
How to create temporary build settings without having to modify a project file.

See also

Visual Studio Projects - C++
.vcxproj and .props file structure
Property page XML files

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Share or reuse Visual Studio project settings

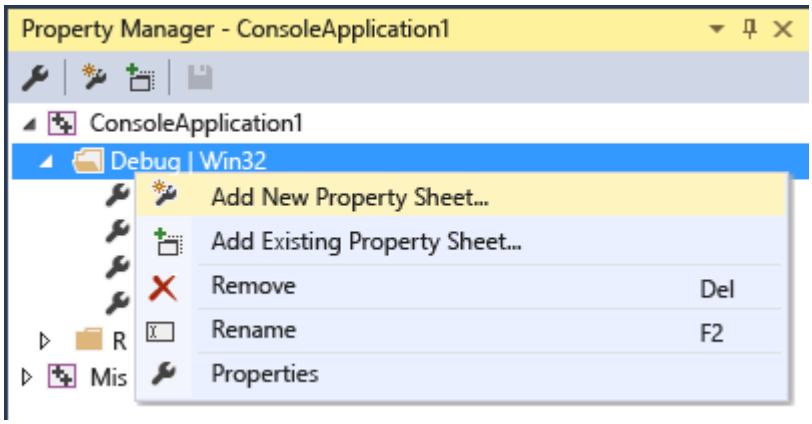
Article • 02/08/2022

To create a custom group of settings that you can share with others or reuse in multiple projects, use **Property Manager** to create a *property sheet* (a `.props` file) to store the settings for each kind of project that you want to be able to reuse or share with others. Using property sheets are far less error-prone than other ways of creating "global" settings.

> **ⓘ Important**
>
> **The problem with `*.user` files**
>
> Past versions of Visual Studio used global property sheets that had a `.user` file name extension and were located in the `\<userprofile>\AppData\Local\Microsoft\MSBuild\v4.0\` folder. We no longer recommend these files because they set properties for project configurations on a per-user, per-computer basis. Such "global" settings can interfere with builds, especially when you are targeting more than one platform on your build computer. For example, if you have both an MFC project and Windows Phone project, the `.user` properties would be invalid for one of them. Reusable property sheets are more flexible and more robust.
>
> Although `.user` files are still installed by Visual Studio and participate in property inheritance, they're empty by default. The best practice is to delete any reference to them in **Property Manager** to ensure that your projects operate independently of any per-user, per-computer settings. This practice is important to ensure correct behavior in a SCC (source code control) environment.

To display **Property Manager**, on the menu bar, choose **View** > **Property Manager** or **View** > **Other Windows** > **Property Manager**, depending on your settings.

If you want to apply a common, frequently used set of properties to multiple projects, you can use **Property Manager** to capture them in a reusable *property sheet* file, which by convention has a `.props` file name extension. You can apply the sheet (or sheets) to new projects so you don't have to set those properties from scratch.



Under each configuration node, you see nodes for each property sheet that applies to that configuration. The system adds property sheets that set common values based on options you choose in the app wizard when you create the project. Right-click any node and choose Properties to see the properties that apply to that node. All the property sheets are imported automatically into the project's primary property sheet (`ms.cpp.props`) and are evaluated in the order they appear in Property Manager. You can move them to change the evaluation order. Property sheets that are evaluated later override the values in previously evaluated sheets. For more information about the order of evaluation in the `.vcxproj` file, the `.props` and `.targets` files, environment variables, and the command line, see Project property inheritance.

If you choose **Add New Project Property Sheet** and then select, for example, the MyProps.props property sheet, a property page dialog box appears. Notice that it applies to the MyProps property sheet; any changes you make are written to the sheet, not to the project file (.vcxproj).

Properties in a property sheet are overridden if the same property is set directly in the `.vcxproj` file.

You can import a property sheet as often as required. Multiple projects in a solution can inherit settings from the same property sheet, and a project can have multiple sheets. A property sheet itself can inherit settings from another property sheet.

You can also create a common property sheet for multiple configurations. To create a property sheet for each configuration, open the shortcut menu for one of them, choose **Add Existing Property Sheet**, and then add the other sheets. However, if you use a common property sheet, properties you set for all configurations that the sheet applies to. The IDE doesn't show which projects or other property sheets inherit from a given property sheet.

In large solutions that have many projects, it can be useful to create a common property sheet for all the projects in the solution. Create the property sheet as usual. Use **Property Manager** to add that property sheet to each project in the solution. If

necessary at the project level, you can add another property sheet to set project-specific values.

> **ⓘ Important**
>
> A `.props` file by default does not participate in source control because it isn't created as a project item. You can manually add the file as a solution item if you want to include it in source control.

To create a property sheet

1. On the menu bar, choose **View** > **Property Manager** or **View** > **Other Windows** > **Property Manager**. The **Property Manager** opens.
2. To define the scope of the property sheet, select the item to which it applies. This item can be a particular configuration, or another property sheet. Open the shortcut menu for this item and then choose **Add New Project Property Sheet**. Specify a name and location.
3. In **Property Manager**, open the new property sheet and then set the properties you want to include.

Property inheritance in Visual Studio projects

Article • 08/03/2021

The Visual Studio native project system is based on MSBuild. MSBuild defines file formats and rules for building projects of any kind. It manages most of the complexity of building for multiple configurations and platforms. You'll find it useful to understand how it works. That's especially important if you want to define custom configurations. Or, to create reusable sets of properties that you can share and import into multiple projects.

The .vcxproj file, .props files and .targets files

Project properties are stored in several files. Some are stored directly in the `.vcxproj` project file. Others come from other `.targets` or `.props` files that the project file imports and which supply default values. You'll find the Visual Studio project files in a locale-specific folder under the base directory, `%VSINSTALLDIR%MSBuild\Microsoft\VC\<version>`. The `<version>` is specific to the version of Visual Studio. It's `v160` for Visual Studio 2019.

Properties are also stored in any custom `.props` files that you might add to your own project. We highly recommend that you *NOT* edit those files manually. Instead, use the property pages in the IDE to modify all properties, especially the ones that participate in inheritance, unless you have a deep understanding of MSBuild and `.vcxproj` files.

As shown earlier, the same property for the same configuration may be assigned a different value in these different files. When you build a project, the MSBuild engine evaluates the project file and all the imported files in a well-defined order that's described later. As each file is evaluated, any property values defined in that file will override the existing values. Any values that aren't specified are inherited from files that were evaluated earlier. When you set a property with property pages, it's also important to pay attention to where you set it. If you set a property to "X" in a `.props` file, but the property is set to "Y" in the project file, then the project will build with the property set to "Y". If the same property is set to "Z" on a project item, such as a `.cpp` file, then the MSBuild engine will use the "Z" value.

Here's the basic inheritance tree:

1. Default settings from the MSBuild CPP Toolset (the *`Microsoft.Cpp.Default.props`* file in the base directory, which is imported by the *`.vcxproj`* file.)
2. Property sheets
3. *`.vcxproj`* file. (This file can override the default and property sheet settings.)
4. Items metadata

> **Tip**
>
> On a property page, a property in **bold** is defined in the current context. A property in normal font is inherited.

View an expanded project file with all imported values

Sometimes it's useful to view the expanded file to determine how a given property value is inherited. To view the expanded version, enter the following command at a Visual Studio command prompt. (Change the placeholder file names to the one you want to use.)

> **msbuild /pp:***temp***.txt** *myapp***.vcxproj**

Expanded project files can be large and difficult to understand unless you're familiar with MSBuild. Here's the basic structure of a project file:

1. Fundamental project properties, which aren't exposed in the IDE.
2. Import of *`Microsoft.cpp.default.props`*, which defines some basic, toolset-independent properties.
3. Global Configuration properties (exposed as **PlatformToolset** and **Project** default properties on the **Configuration General** page. These properties determine which toolset and intrinsic property sheets are imported in *`Microsoft.cpp.props`* in the next step.
4. Import of *`Microsoft.cpp.props`*, which sets most of the project defaults.
5. Import of all property sheets, including *`.user`* files. These property sheets can override everything except the **PlatformToolset** and **Project** default properties.

6. The rest of the project configuration properties. These values can override what was set in the property sheets.

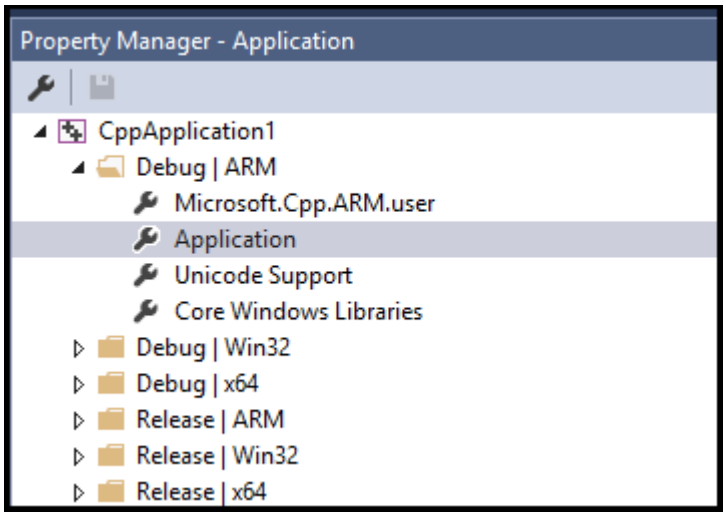
7. Items (files) together with their metadata. These items are always the last word in MSBuild evaluation rules, even if they occur before other properties and imports.

For more information, see MSBuild Properties.

Build configurations

A configuration is just an arbitrary group of properties that are given a name. Visual Studio provides Debug and Release configurations. Each sets various properties appropriately for a debug build or release build. You can use the **Configuration Manager** to define custom configurations. They're a convenient way to group properties for a specific flavor of build.

To get a better idea of build configurations, open **Property Manager**. You can open it by choosing **View** > **Property Manager** or **View** > **Other Windows** > **Property Manager**, depending on your settings. **Property Manager** has nodes for each configuration and platform pair in the project. Under each of these nodes are nodes for property sheets (`.props` files) that set some specific properties for that configuration.



For example, you can go to the General pane in the Property Pages. Change the Character Set property to "Not Set" instead of "Use Unicode", and then click **OK**. The Property Manager now shows no **Unicode Support** property sheet. It's removed for the current configuration, but it's still there for other configurations.

For more information about Property Manager and property sheets, see Share or reuse Visual Studio C++ project settings.

> **Tip**

The `.user` file is a legacy feature. We recommend that you delete it, to keep properties correctly grouped according to configuration and platform.

How to: Modify C++ project properties and targets without changing the project file

Article • 07/28/2023

You can override project properties and targets from the MSBuild command prompt without changing the project file. This is useful when you want to apply some properties temporarily or occasionally. It assumes some knowledge of MSBuild. For more information, see MSBuild.

> **Important**
>
> You can use the XML Editor in Visual Studio, or any text editor, to create the .props or .targets file. Don't use the **Property Manager** in this scenario because it adds the properties to the project file.

To override project properties:

1. Create a `.props` file that specifies the properties you want to override.
2. From the command prompt: `set ForceImportBeforeCppTargets="C:\sources\my_props.props"`

To override project targets:

1. Create a `.targets` file with their implementation or a particular target
2. From the command prompt: `set ForceImportAfterCppTargets="C:\sources\my_target.targets"`

You can also set either option on the msbuild command line by using the `/p:` option:

Windows Command Prompt

```
msbuild myproject.sln
/p:ForceImportBeforeCppTargets="C:\sources\my_props.props"
msbuild myproject.sln
/p:ForceImportAfterCppTargets="C:\sources\my_target.targets"
```

Overriding properties and targets in this way is equivalent to adding the following imports to all `.vcxproj` files in the solution:

XML

```xml
<Import Project="C:\sources\my_props.props" />
<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />
<Import Project="C:\sources\my_target.targets" />
```

Clang/LLVM support in Visual Studio projects

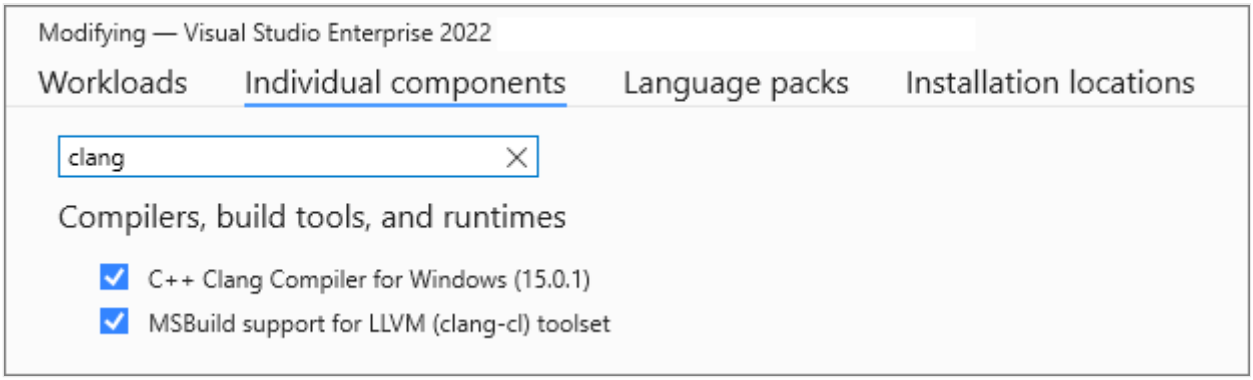
Article • 03/20/2024

You can use Visual Studio 2019 version 16.2 and later with Clang/LLVM to edit, build, and debug C++ Visual Studio projects (MSBuild) that target Windows or Linux.

Install

For the best IDE support in Visual Studio, we recommend using the latest Clang compiler tools for Windows. If you don't already have the tools, you can install them by opening the Visual Studio Installer and choosing **C++ Clang tools for Windows** under **Desktop development with C++** optional components. You may prefer to use an existing Clang installation on your machine; if so, choose **MSBuild support for LLVM (clang-cl) toolset**.

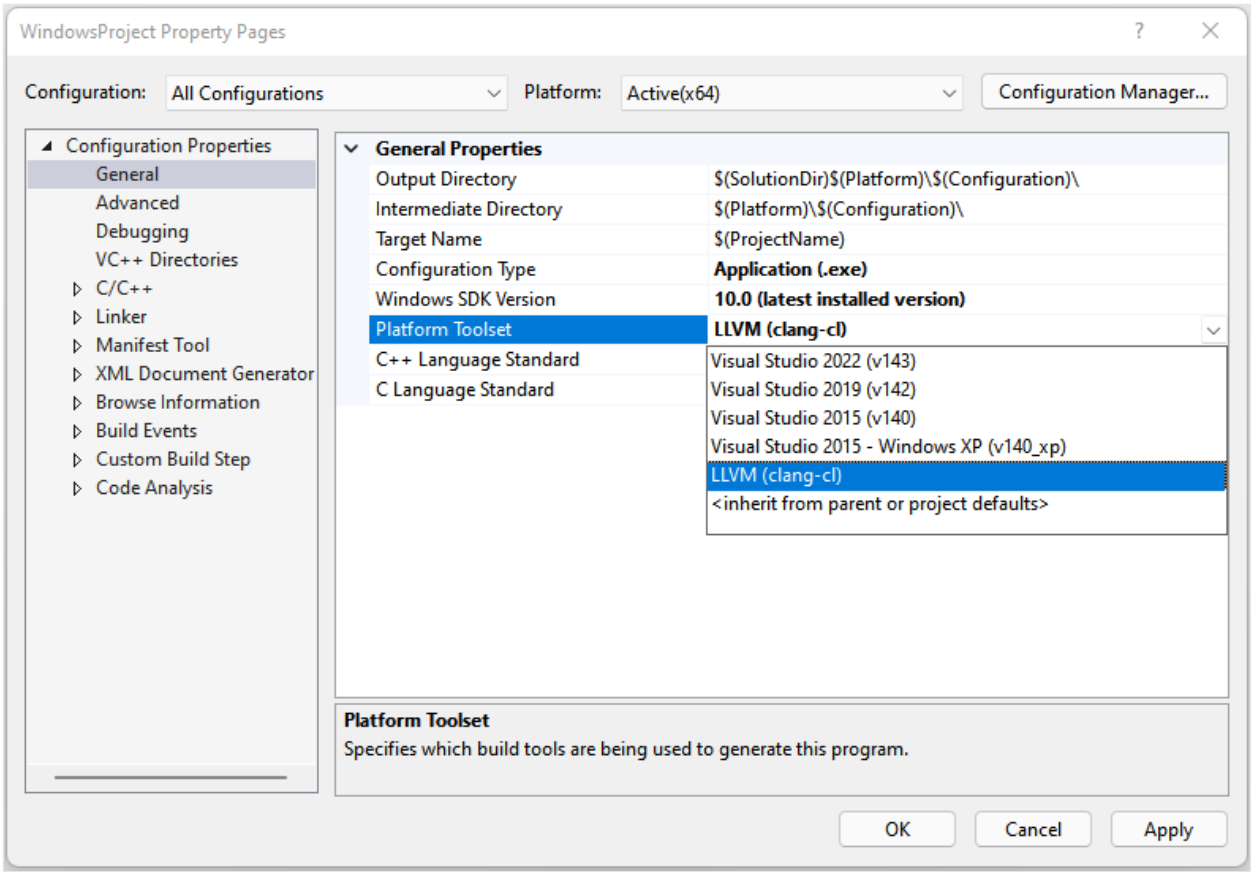
The Microsoft C++ Standard Library requires at least Clang 8.0.0.



Later versions of Visual Studio provide newer versions of the Clang toolset. The bundled version of Clang gets updated automatically to stay current with updates in the Microsoft implementation of the Standard Library. For example, Visual Studio 2019 version 16.11 includes Clang v12.

Configure a Windows project to use Clang tools

To configure a Visual Studio project to use Clang, right-click on the project node in **Solution Explorer** and choose **Properties**. Typically, you should first choose **All configurations** at the top of the dialog. Then, under **General** > **Platform Toolset**, choose **LLVM (clang-cl)** and then **OK**.



If you're using the Clang tools that are bundled with Visual Studio, no extra steps are required. For Windows projects, Visual Studio by default invokes Clang in clang-cl mode. It links with the Microsoft implementation of the Standard Library. By default, **clang-cl.exe** is located in `*%VCINSTALLDIR%\Tools\Llvm\bin\*` and `*%VCINSTALLDIR%\Tools\Llvm\x64\bin\*`.

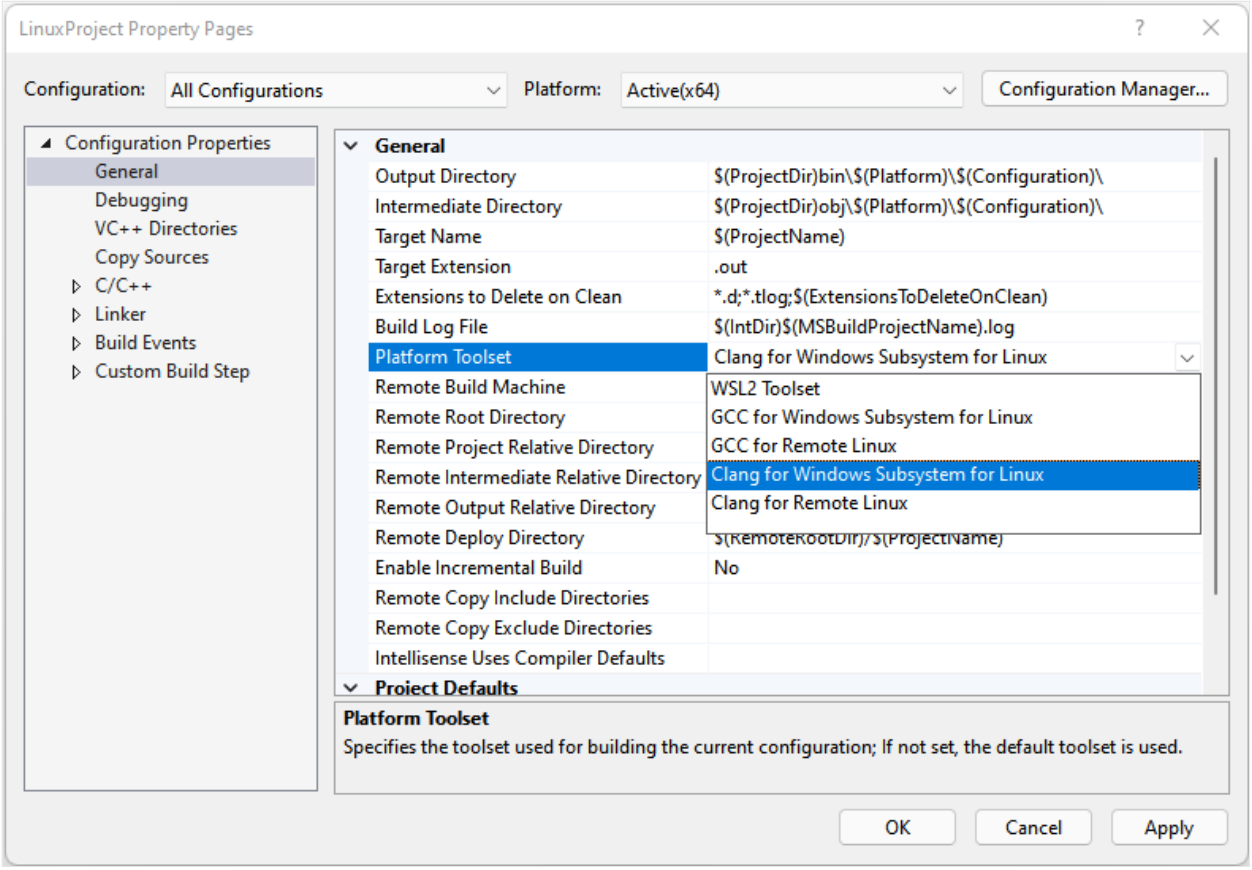
If you're using a custom Clang installation, you can change the value of the `LLVMInstallDir` property. For more information, see Set a custom LLVM location.

Configure a Linux project to use Clang tools

For Linux projects, Visual Studio uses the Clang GCC-compatible frontend. The project properties and nearly all compiler flags are identical

To configure a Visual Studio Linux project to use Clang:

1. Right-click on the project node in **Solution Explorer** and choose **Properties**.
2. Typically, you should first choose **All configurations** at the top of the dialog.
3. Under **General** > **Platform Toolset**, choose **Clang for Windows Subsystem for Linux** if you're using Windows Subsystem for Linux (WSL). Choose **Clang for Remote Linux** if you're using a remote machine or VM.
4. Press **OK**.



On Linux, Visual Studio by default uses the first Clang location that it finds in the PATH environment property. If you're using a custom Clang installation, then either change the value of the `LLVMInstallDir` property or else enter the path under **Project** > **Properties** > **Configuration Properties** > **VC++ DIrectories** > **Executable Directories**. For more information, see Set a custom LLVM location.

Set a custom LLVM location and toolset

To set a custom path to LLVM and set a custom LLVM toolset version for one or more projects, create a *Directory.build.props* file. Then, add that file to the root folder of any project. You can add it to the root solution folder to apply it to all projects in the solution. The file should look like this example (but use your actual LLVM path and version number):

XML

```
<Project>
  <PropertyGroup>
    <LLVMInstallDir>C:\MyLLVMRootDir</LLVMInstallDir>
    <LLVMToolsVersion>15.0.0</LLVMToolsVersion>
  </PropertyGroup>
</Project>
```

Set a custom LLVM toolset version in the IDE

Starting in Visual Studio 2019 version 16.9, you can set a custom toolset version for LLVM in Visual Studio. To set this property in a project:

1. Open the project's **Property Pages** dialog box. For more information, see Set C++ compiler and build properties.
2. Select the **Configuration Properties** > **General** property page.
3. Modify the **Platform Toolset** property to *LLVM (clang-cl)*, if it isn't already set. Choose **Apply** to save your changes.
4. Select the **Configuration Properties** > **Advanced** property page.
5. Modify the **LLVM Toolset Version** property to your preferred version, and then choose **OK** to save your changes.

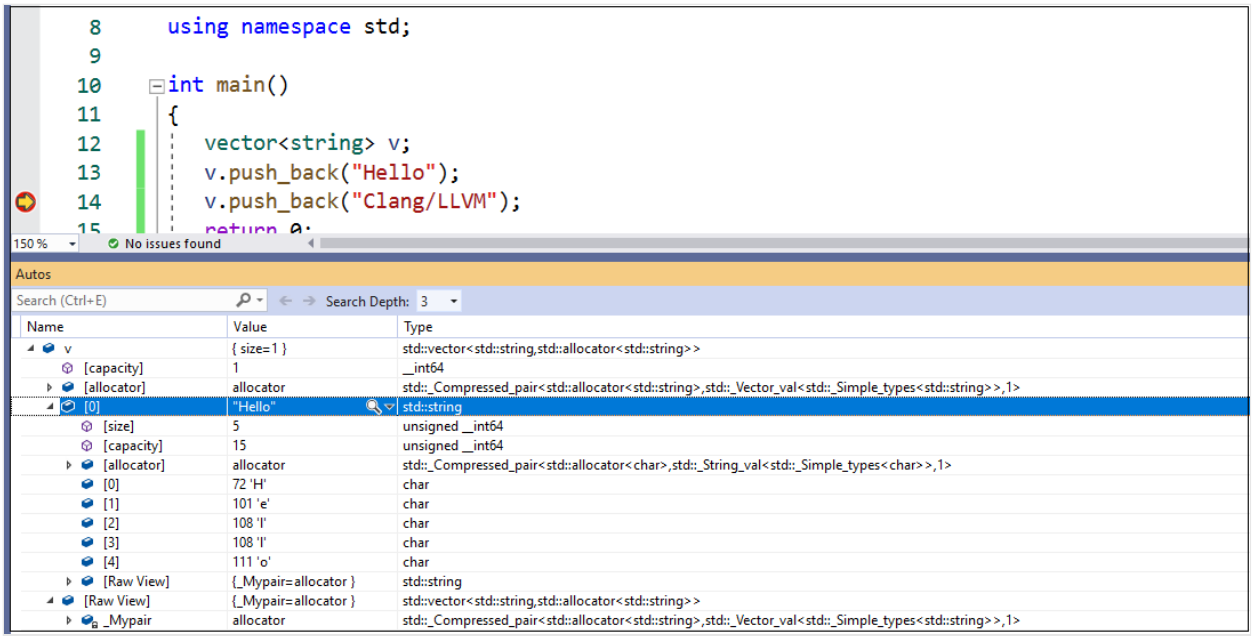
The **LLVM Toolset Version** property only appears when the LLVM platform toolset is selected.

When you add a `Directory.build.props` file to a project or solution, the settings appear as the default in the project Property Pages dialog. However, changes to these properties in Visual Studio override the settings in the `Directory.build.props` file.

Set properties, edit, build, and debug

After you have set up a Clang configuration, right-click again on the project node and choose **Reload project**. You can now build and debug the project using the Clang tools. Visual Studio detects that you're using the Clang compiler and provides IntelliSense, highlighting, navigation, and other editing features. Errors and warnings are displayed in the **Output Window**. The project property pages for a Clang configuration are similar to the ones for MSVC. However, some compiler-dependent features such as Edit and Continue aren't available for Clang configurations. You can set a Clang compiler or linker option that isn't available in the property pages. Add it manually in the property pages under **Configuration Properties** > **C/C++ (or Linker)** > **Command Line** > **Additional Options**.

When debugging, you can use breakpoints, memory and data visualization, and most other debugging features.



Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Consuming libraries and components

Article • 03/02/2022

C++ projects often need to call functions or access data in a binary file such as static library (LIB files), DLL, Windows Runtime component, COM component, or .NET assembly. In these cases, you have to configure the project so that it can find that binary at build time. The specific steps depend on the type of your project, the type of the binary, and whether the binary gets built in the same solution as your project.

Consuming libraries downloaded via vcpkg

To consume a library that you have downloaded by using the **vcpkg** package manager, you can ignore the instructions below. For more information, see vcpkg.io.

Consuming static libraries

If your static library project gets built in the same solution:

1. #include the header file(s) for the static library using quotation marks. In a typical solution, the path starts with `../<library project name>`. IntelliSense will help you find it.
2. Add a reference to the static library project. Right-click on **References** under the application project node in **Solution Explorer** and choose **Add Reference**.

If the static library isn't part of the solution:

1. Right-click on the application project node in **Solution Explorer** and then choose **Properties**.
2. In the **VC++ Directories** property page, add the path to the directory that contains the LIB file to **Library Paths**. Then, add the path to the library header file(s) to **Include Directories**.
3. In the **Linker** > **Input** property page, add the name of the LIB file to **Additional Dependencies**.

Dynamic link libraries

If the DLL gets built as part of the same solution as the application, follow the same steps as for a static library.

If the DLL isn't part of the application solution, you need: the DLL file, the header(s) with prototypes for the exported functions and classes, and a LIB file that provides the necessary linking information.

1. Copy the DLL to the output folder of your project, or to another folder in the standard Windows search path for DLLs. For more information, see Dynamic-Link Library Search Order.
2. Follow steps 1-3 for static libraries to provide the paths to the headers and LIB file.

COM objects

If your native C++ application needs to consume a COM object, and that object is *registered*, then all you have to do is call CoCreateInstance and pass in the CLSID of the object. The system will find it in the Windows Registry and load it. A C++/CLI project can consume a COM object in the same way. Or, it can consume it by adding a reference to it from the **Add References** > **COM** list and consuming it through its Runtime callable wrapper.

.NET assemblies and Windows Runtime Components

In UWP or C++/CLI projects, you consume .NET assemblies or Windows Runtime Components by adding a *reference* to the assembly or component. Under the **References** node in a UWP or C++/CLI project, you see references to commonly used components. Right-click on the **References** node in **Solution Explorer** to bring up the **Reference Manager** and browse through the components available on the system. Choose the **Browse** button to navigate to any folder that contains a custom component. Because .NET assemblies and Windows Runtime components contain built-in type information, you can view their methods and classes by right-clicking and choosing **View in Object Browser**.

Reference properties

Each kind of reference has properties. You can view the properties by selecting the reference in Solution Explorer and pressing **Alt + Enter**, or else right-clicking and choosing **Properties**. Some properties are read-only and some are modifiable. However, typically you don't have to manually modify these properties.

ActiveX reference properties

ActiveX reference properties are available only for references to COM components. These properties get displayed only when you select a COM component in the **References** pane. The properties aren't modifiable.

- **Control Full Path**

 Displays the directory path of the referenced control.

- **Control GUID**

 Displays the GUID for the ActiveX control.

- **Control Version**

 Displays the version of the referenced ActiveX control.

- **Type Library Name**

 Displays the name of the referenced type library.

- **Wrapper Tool**

 Displays the tool that's used to build the interop assembly from the referenced COM library or ActiveX control.

Assembly reference properties (C++/CLI)

Assembly reference properties are available only for references to .NET Framework assemblies in C++/CLI projects. These properties get displayed only when you select a .NET Framework assembly in the **References** pane. The properties aren't modifiable.

- **Relative Path**

 Displays the relative path from the project directory to the referenced assembly.

Build properties

The following properties are available on various kinds of references. They enable you to specify how to build with references.

- **Copy Local**

 Specifies whether to automatically copy the referenced assembly to the target location during a build.

- **Copy Local Satellite Assemblies (C++/CLI)**

Specifies whether to automatically copy the satellite assemblies of the referenced assembly to the target location during a build. Only used if **Copy Local** is `true`.

- **Reference Assembly Output**

 Specifies that this assembly gets used in the build process. If `true`, the assembly gets used on the compiler command line during the build.

Project-to-project reference properties

The following properties define a *project-to-project reference* from the project that's selected in the **References** pane to another project in the same solution. For more information, see Managing references in a project.

- **Link Library Dependencies**

 When this property is **True**, the project system links the LIB files that the independent project produces into the dependent project. Typically, you'll specify **True**.

- **Project Identifier**

 Uniquely identifies the independent project. The property value is an internal system GUID that isn't modifiable.

- **Use Library Dependency Inputs**

 When this property is **False**, the project system won't link the OBJ files for the library that the independent project produces into the dependent project. That's why this value disables incremental linking. Typically, you'll specify **False** because building the application can take a long time if there are many independent projects.

Read-only reference properties (COM & .NET)

The following properties exist on COM and .NET assembly references, and aren't modifiable.

- **Assembly Name**

 Displays the assembly name for the referenced assembly.

- **Culture**

 Displays the culture of the selected reference.

- **Description**

 Displays the description of the selected reference.

- **Full Path**

 Displays the directory path of the referenced assembly.

- **Identity**

 For the .NET Framework assemblies, displays the full path. For COM components, displays the GUID.

- **Label**

 Displays the label of the reference.

- **Name**

 Displays the name of the reference.

- **Public Key Token**

 Displays the public key token used to identify the referenced assembly.

- **Strong Name**

 `true` if the referenced assembly has a strong name. A strong named assembly has a unique version.

- **Version**

 Displays the version of the referenced assembly.

See also

C++ project property page reference
Set C++ compiler and build properties in Visual Studio

How to: Organize Project Output Files for Builds

Article • 08/03/2021

This topic describes best practices for organizing project output files. Build errors can occur when you set up project output files incorrectly. This topic also outlines the advantages and disadvantages of each alternative for organizing your project output files.

Referencing CLR Assemblies

To reference assemblies with #using

1. You can reference an assembly directly from your code by using the #using directive, such as `#using <System.Data.dll>`. For more information, see #using Directive.

 The file specified can be a .dll, .exe, .netmodule, or .obj, as long as it is in MSIL. The referenced component can be built in any language. Using this option, you will have access to IntelliSense since the metadata will be extracted from the MSIL. The file in question must be in the path for the project; otherwise, the project will not compile and IntelliSense will not be available. An easy way to determine whether the file is in the path is to right-click on the #using line and choose the **Open document** command. You will be notified if the file cannot be found.

 If you do not want to put the full path to the file, you can use the **/AI** compiler option to edit the search path for #using references. For more information, see /AI (Specify Metadata Directories).

To reference assemblies with /FU

1. Instead of referencing an assembly directly from a code file as described above, you can use the **/FU** compiler option. The advantage to this method is that you do not have to add a separate #using statement to every file that references a given assembly.

 To set this option, open the **Properties Pages** for the project. Expand the **Configuration Properties** node, and then expand the **C/C++** node and select

Advanced. Add the desired assemblies next to **Force #using**. For more information, see /FU (Name Forced #using File).

To reference assemblies with Add New Reference

1. This is the easiest way to use CLR assemblies. First, make sure the project is compiled with the **/clr** compiler option. Then, right click the project from the **Solution Explorer** and select **Add**, **References**. The **Property Pages** dialog will appear.

2. From the **Property Pages** dialog, select **Add New Reference**. A dialog will appear listing all .NET, COM, and other assemblies available in the current project. Select the desired assembly and click **OK**.

 Once a project reference is set, the corresponding dependencies are automatically handled. In addition, since metadata is part of an assembly, there is no need to add a header file or prototype the elements that are being used from managed assemblies.

Referencing Native DLLs or Static Libraries

To reference native DLLs or static libraries

1. Reference the appropriate header file in your code using the #include directive. The header file must be in the include path or part of the current project. For more information, see #include Directive (C/C++).

2. You can also set project dependencies. Setting project dependencies guarantees two things. First, it ensures that projects are built in the right order so that a project can always find the dependent files it needs. Second, it implicitly adds the dependent project's output directory to the path so that files can be found easily at link-time.

3. To deploy the application, you will need to place the DLL in an appropriate place. This can be one of the following:

 a. The same path as the executable.

 b. Anywhere in the system path (the **path** environment variable).

 c. In the side-by-side assembly. For more information, see Building C/C++ Side-by-side Assemblies.

Working with Multiple Projects

By default, projects are built such that all output files are created in a subdirectory of the project directory. The directory is named based on the build configuration (e.g. Debug or Release). In order for sibling projects to refer to each other, each project must explicitly add the other project output directories to their path in order for linking to succeed. This is done automatically when you set the project dependencies. However, if you do not use dependencies, you must carefully handle this because builds can become very difficult to manage. For example, when a project has Debug and Release configurations, and it includes an external library from a sibling project, it should use a different library file depending on which configuration is being built. Thus, hard-coding these paths can be tricky.

All essential output files (such as executables, incremental linker files, and PDB files) are copied into a common solution directory. Thus, when working with a solution that contains a number of C++ projects with equivalent configurations, all the output files are centralized for simplified linking and deployment. You can be sure that their application/library will work as expected if they keep those files together (since the files are guaranteed to be in the path).

The location of output files can be a major issue when deploying to a production environment. While running projects in the IDE, the paths to included libraries are not necessarily the same as in the production environment. For example, if you have `#using "../../lib/debug/mylib.dll"` in your code but then deploy mylib.dll into a different relative position, the application will fail at runtime. To prevent this, you should avoid using relative paths in #include statements in your code. It is better to ensure that the necessary files are in the project build path and similarly ensuring that the corresponding production files are properly placed.

How to specify where output files go

1. The location of project output settings can be found in the project's **Property Pages**. Expand the node next to **Configuration Properties** and select **General**. The output location is specified next to **Output Directory**. For more information, see General Property Page (Project).

See also

C++ project types in Visual Studio

Understanding Custom Build Steps and Build Events

Article • 08/03/2021

From within the Visual C++ development environment, there are three basic ways to customize the build process:

- **Custom Build Steps**

 A custom build step is a build rule associated with a project. A custom build step can specify a command line to execute, any additional input or output files, and a message to display. For more information, see How to: Add a Custom Build Step to MSBuild Projects.

- **Custom Build Tools**

 A custom build tool is a build rule associated with one or more files. A custom build step can pass input files to a custom build tool, which results in one or more output files. For example, the help files in an MFC application are built with a custom build tool. For more information, see How to: Add Custom Build Tools to MSBuild Projects and Specifying Custom Build Tools.

- **Build Events**

 Build events let you customize a project's build. There are three build events: *pre-build*, *pre-link*, and *post-build*. A build event lets you specify an action to occur at a specific time in the build process. For example, you could use a build event to register a file with **regsvr32.exe** after the project finishes building. For more information, see Specifying Build Events.

Troubleshooting Build Customizations can help you ensure that your custom build steps and build events run as expected.

The output format of a custom build step or build event can also enhance the usability of the tool. For more information, see Formatting the Output of a Custom Build Step or Build Event.

For each project in a solution, build events and custom build steps run in the following order along with other build steps:

1. Pre-Build event

2. Custom build tools on individual files

3. MIDL
4. Resource compiler
5. The C/C++ compiler
6. Pre-Link event
7. Linker or Librarian (as appropriate)
8. Manifest Tool
9. BSCMake
10. Custom build step on the project
11. Post-Build event

The `custom build step on the project` and a `post-build event` run sequentially after all other build processes finish.

In this section

Specify Custom Build Tools
Specify Build Events
Troubleshoot Build Customizations
Format the Output of a Custom Build Step or Build Event

See also

Visual Studio Projects - C++
Common macros for build commands and properties

Specify custom build tools

Article • 08/03/2021

A *custom build tool* provides the build system with the information it needs to build specific input files. A custom build tool specifies a command to run, a list of input files, a list of output files that are generated by the command, and an optional description of the tool.

For general information about custom build tools and custom build steps, see Understanding Custom Build Steps and Build Events.

To specify a custom build tool

1. Open the project's **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.

2. Choose **Configuration Properties** to enable the **Configuration** box. In the **Configuration** box, select the configuration for which you want to specify a custom build tool.

3. In **Solution Explorer**, select the input file for the custom build tool.

 If the **Custom Build Tool** folder does not appear, the file extension of the file you selected is associated with a default tool. For example, the default tool for .c and .cpp files is the compiler. To override a default tool setting, in the **Configuration Properties** node, in the **General** folder, in the **Item Type** property, choose **Custom Build Tool**. Choose **Apply** and the **Custom Build Tool** node is displayed.

4. In the **Custom Build Tool** node, in the **General** folder, specify the properties associated with the custom build tool:

 - In **Additional Dependencies**, specify any additional files beyond the one for which the custom build tool is being defined (the file associated with the custom build tool is implicitly considered an input to the tool). Having additional input files is not a requirement for a custom build tool. If you have more than one additional input, separate them with semicolons.

 If an **Additional Dependencies** file's date is later than the input file, then the custom build tool is run. If all of the **Additional Dependencies** files are older than the input file, and the input file is older than the **Outputs** property file, then the custom build tool is not run.

For example, suppose you have a custom build tool that takes MyInput.x as input and generates MyInput.cpp, and that MyInput.x includes a header file, MyHeader.h. You can specify MyHeader.h as an input dependency to MyInput.x, and the build system will build MyInput.cpp when it is out-of-date with respect to MyInput.x or MyHeader.h.

Input dependencies can also ensure that your custom build tools run in the order you need them to. In the preceding example, suppose that MyHeader.h is actually the output of a custom build tool. Because MyHeader.h is a dependency of MyInput.x, the build system will first build Myheader.h before running the custom build tool on MyInput.x.

- In **Command Line**, specify a command as if you were specifying it at the command prompt. Specify a valid command or batch file, and any required input or output files. Specify the **call** batch command before the name of a batch file to guarantee that all subsequent commands are executed.

 Multiple input and output files can be specified symbolically with MSBuild macros. For information on how to specify the location of files, or the names of sets of files, see Common macros for build commands and properties.

 Because the '%' character is reserved by MSBuild, if you specify an environment variable replace each % escape character with the **%25** hexadecimal escape sequence. For example, replace **%WINDIR%** with **%25WINDIR%25**. MSBuild replaces each **%25** sequence with the % character before it accesses the environment variable.

- In **Description**, enter a descriptive message about this custom build tool. The message is printed to the **Output** window when the build system processes this tool.

- In **Outputs**, specify the name of the output file. This is a required entry; without a value for this property, the custom build tool will not run. If a custom build tool has more than one output, separate file names with a semicolon.

 The name of the output file should be the same as it is specified in the **Command Line** property. The project build system will look for the file and check its date. If the output file is older than the input file or if the output file is not found, the custom build tool is run. If all of the **Additional Dependencies** files are older than the input file, and the input file is older than the file specified in the **Outputs** property, the custom build tool is not run.

If you want the build system to operate on an output file generated by the custom build tool, you must manually add it to the project. The custom build tool will update the file during the build.

Example

Assume that you want to include a file named parser.l in your project. You have a lexical analyzer, **lexer.exe**, on your executable path. You want to use it to process parser.l to produce a .c file that has the same base name (parser.c).

First, add parser.l and parser.c to the project. If the files do not yet exist, add a reference to the files. Create a custom build tool for parser.l and enter the following in the **Commands** property:

> **lexer %(FullPath) .%(Filename).c**

This command runs the lexical analyzer on parser.l and outputs parser.c to the project directory.

In the **Outputs** property, enter the following:

> **.%(Filename).c**

When you build the project, the build system compares the timestamps of parser.l and parser.c. If parser.l is more recent, or if parser.c doesn't exist, the build system runs the value of the **Command Line** property to bring parser.c up to date. Since parser.c was also added to the project, the build system then compiles parser.c.

See also

Common macros for build commands and properties
Troubleshooting Build Customizations

Specifying build events

Article • 08/03/2021

You can use build events to specify commands that run before the build starts, before the link process, or after the build finishes.

Build events are executed only if the build successfully reaches those points in the build process. If an error occurs in the build, the *post-build* event does not occur; if the error occurs before the linking phase, neither the *pre-link* nor the *post-build* event occurs. Additionally, if no files need to be linked, the *pre-link* event does not occur. The *pre-link* event is also not available in projects that do not contain a link step.

If no files need to be built, no build events occur.

For general information on build events, see Understanding Custom Build Steps and Build Events.

To specify a build event

1. In **Solution Explorer**, select the project for which you want to specify the build event.

2. Open the project's **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.

3. In the **Build Events** folder, select a build event property page.

4. Specify the properties associated with the build event:

 - In **Command Line**, specify a command as if you were specifying it at the command prompt. Specify a valid command or batch file, and any required input or output files. Specify the **call** batch command before the name of a batch file to guarantee that all subsequent commands are executed.

 Multiple input and output files can be specified symbolically with MSBuild macros. For information on how to specify the location of files, or the names of sets of files, see Common macros for build commands and properties.

 Because the '%' character is reserved by MSBuild, if you specify an environment variable replace each % escape character with the **%25** hexadecimal escape sequence. For example, replace **%WINDIR%** with **%25WINDIR%25**. MSBuild replaces each **%25** sequence with the % character before it accesses the environment variable.

- In **Description**, type a description for this event. The description is printed to the **Output** window when this event occurs.
- In **Excluded From Build**, specify **Yes** if you do not want the event to run.

See also

Understanding Custom Build Steps and Build Events
Common macros for build commands and properties
Troubleshooting Build Customizations

Troubleshooting Build Customizations

Article • 08/03/2021

If your custom build steps or events are not behaving as you expect, there are several things you can do to try to understand what is going wrong.

- Make sure that the files your custom build steps generate match the files you declare as outputs.
- If your custom build steps generate any files that are inputs or dependencies of other build steps (custom or otherwise), make sure that those files are added to your project. And make sure that the tools that consume those files execute after the custom build step.
- To display what your custom build step is actually doing, add `@echo on` as the first command. The build events and build steps are put in a temporary .bat file and run when the project is built. Therefore, you can add error checking to your build event or build step commands.
- Examine the build log in the intermediate files directory to see what actually executed. The path and name of the build log is represented by the **MSBuild** macro expression, **$(IntDir)\$(MSBuildProjectName).log**.
- Modify your project settings to collect more than the default amount of information in the build log. On the **Tools** menu, click **Options**. In the **Options** dialog box, click the **Projects and Solutions** node and then click the **Build and Run** node. Then, in the **MSBuild project build log file verbosity** box, click **Detailed**.
- Verify the values of any file name or directory macros you are using. You can echo macros individually, or you can add `copy %0 command.bat` to the start of your custom build step, which will copy your custom build step's commands to command.bat with all macros expanded.
- Run custom build steps and build events individually to check their behavior.

See also

Understanding Custom Build Steps and Build Events

Formatting the output of a custom build step or build event

Article • 03/22/2022

If the output of a custom build step or build event is formatted correctly, users get the following benefits:

- Warnings and errors are counted in the **Output** window.
- Output appears in the **Task List** window.
- Clicking on the output in the **Output** window displays the appropriate location.
- **F1** operations are enabled in the **Task List** window or **Output** window.

Output format

The format of the output should be:

> { *filename* `(` *line-number* [`,` *column-number*] `)` | *tool-name* } `:` [*any-text*] {`error` | `warning`} *code-type-and-number* `:` *localizable-string* [*any-text*]

Where:

- { *a* | *b* } is a choice of either *a* or *b*,
- [*item*] is an optional string or parameter,
- `text` represents a literal.

For example:

> C:\sourcefile.cpp(134) : error C2143: syntax error : missing ';' before '}'
>
> LINK : fatal error LNK1104: cannot open file 'some-library.lib'

See also

Understanding custom build steps and build events

How to: Create a C++ Project from Existing Code

Article • 08/13/2024

In Visual Studio, you can port existing code files into a C++ project using the **Create New Project From Existing Code Files** wizard. This wizard creates a project solution that uses the MSBuild system to manage source files and build configuration. It works best with relatively simple projects that don't have complex folder hierarchies. The wizard isn't available in older Express editions of Visual Studio.

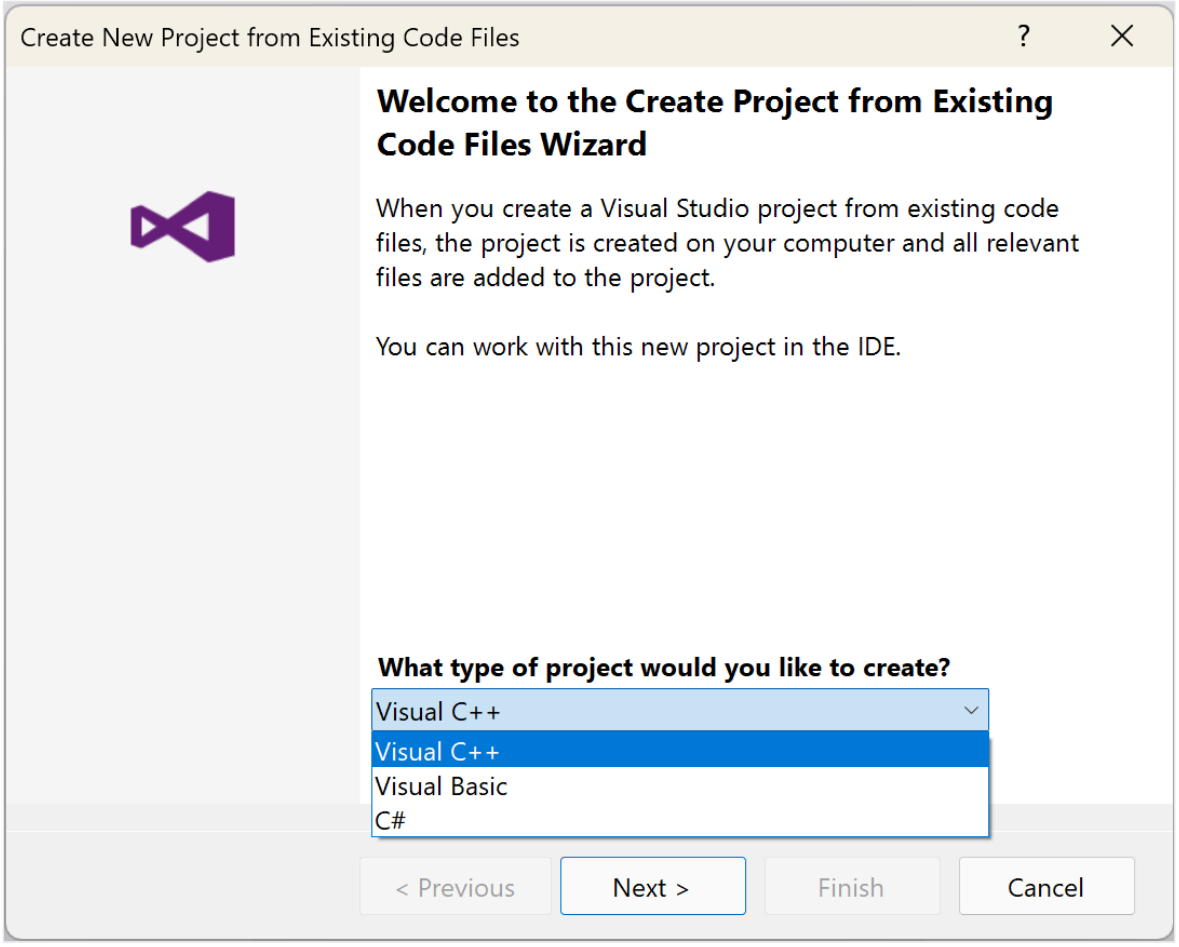
Porting existing code files into a C++ project enables the use of native MSBuild project management features built into the IDE. If you prefer to use your existing build system, such as nmake makefiles, CMake, or alternatives, you can use the Open Folder or CMake options instead. For more information, see Open Folder projects for C++ or CMake projects in Visual Studio. Both options let you use IDE features such as IntelliSense and Project Properties.

To create a C++ project from existing code

The following instructions assume that Visual Studio is running and is past the start page. If you are on the Visual Studio start page, choose **Continue without code** to open the IDE.

1. On the **File** menu, select **New** > **Project From Existing Code**.

2. The **Create New Project from Existing Code Files** wizard opens. Choose what type of project to create from the dropdown: **Visual C++**, **Visual Basic**, or **C#**. Then choose **Next** to continue.

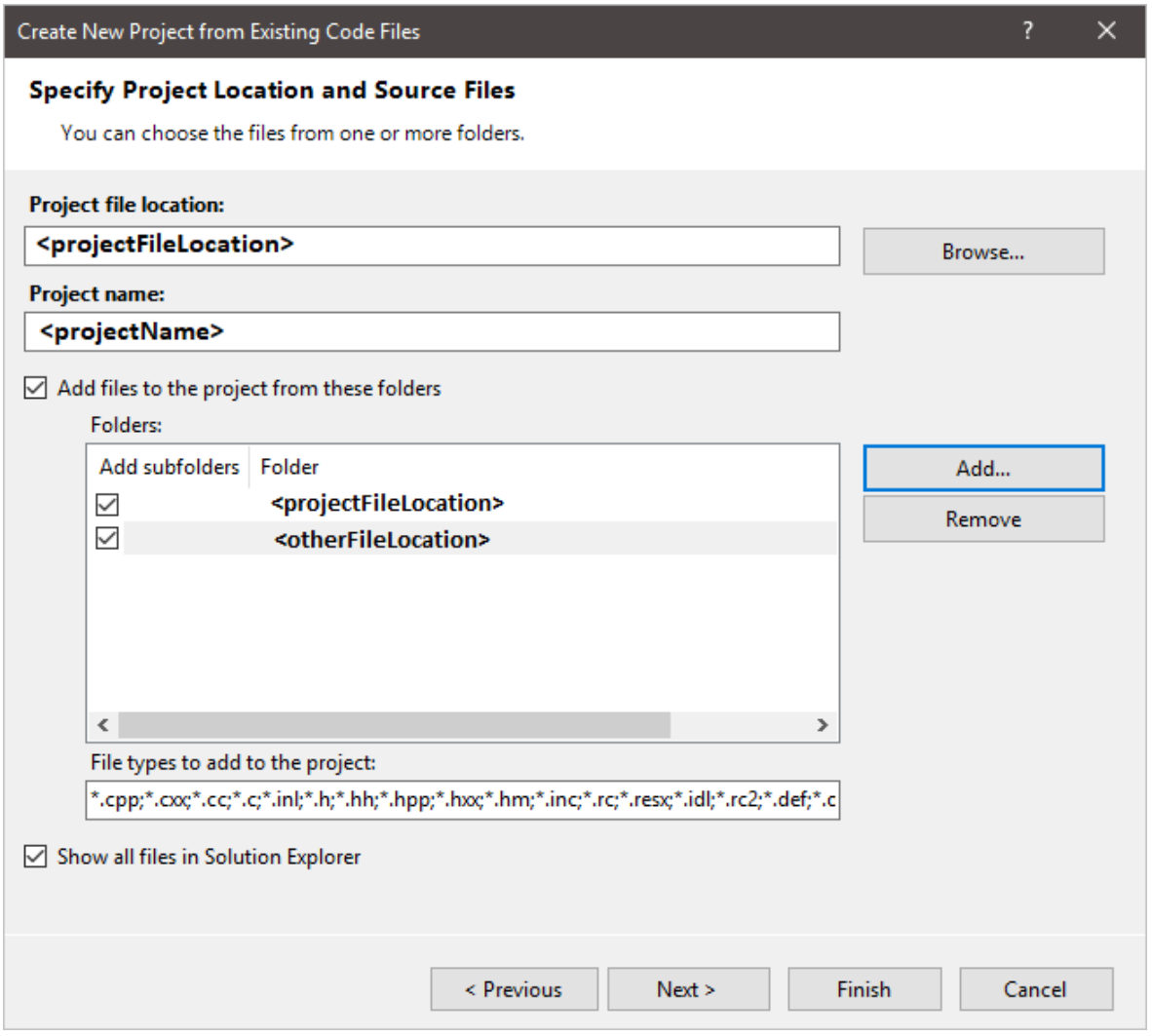


3. Specify your project location, the directory for your source files, and the kinds of files the wizard imports into the new project. Choose **Next** to continue.

Expand table

Setting	Description
Project file location	Specifies the directory path of the new project. This location is where the wizard deposits all the files (and subdirectories) of the new project. Select **Browse** to display the **Project file location** dialog. Navigate to the right folder and specify the directory that contains the new project.
Project name	Specifies the name of the new project. Project files, which have file extensions such as .vcxproj adopts this name and existing code files keep their original name.
Add files to the project from these folders	Check to set the wizard to copy existing code files from their original directories (that are specified in the list box below this control) into the new project. Check **Add Subfolders** to specify copying code files from all subdirectories into the project. The directories are listed in the **Folder** column. - Select **Add** to display the **Add files to the project from this folder** dialog box, to specify directories the wizard searches for existing code

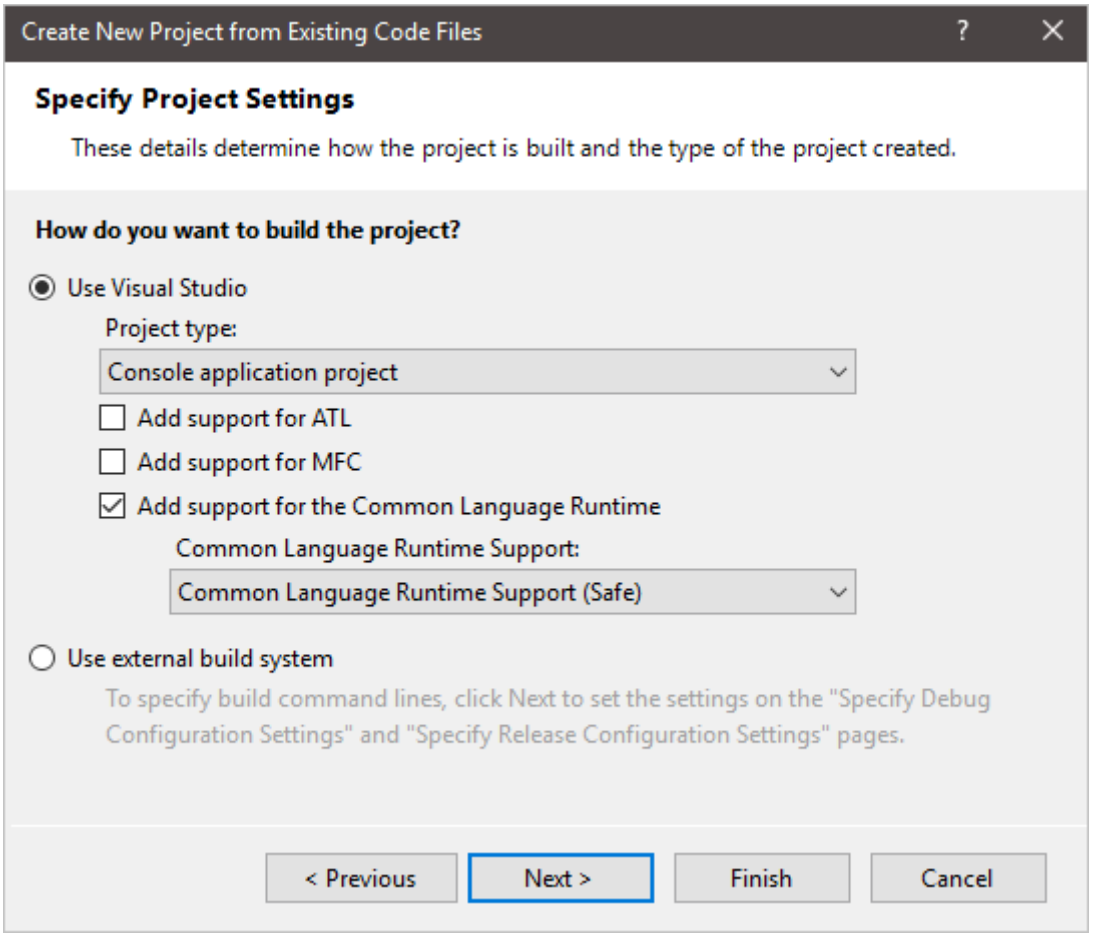
Setting	Description
	files. - Select **Remove** to delete the directory path selected in the list box. In the **File types to add to the project** box, specify the kinds of files that the wizard adds to the new project based on the given file extensions. File extensions are preceded with the asterisk wildcard character and are delimited in the list of file extensions by a semicolon.
Show all files in Solution Explorer	Specifies that all files in the new project to be visible and displayed in the **Solution Explorer** window. This option is enabled by default.



4. Specify the project settings to use such as the build environment for the new project and the build settings to match a specific type of new project to generate. Choose **Next** to continue.

Expand table

Setting	Description
Use Visual Studio	Specifies to use build tools that are included in Visual Studio for building the new project. This option is selected by default. Select **Project Type** to specify the type of project the wizard generates. Choose **Windows application project**, **Console application project**, **Dynamically linked library (DLL) project**, or **Static library (LIB) project**. Check **Add support for ATL** to add ATL support to the new project. Check **Add support for MFC** to add MFC support to the new project. Check **Add support for the Common Language Runtime** to add CLR programming support to the project. Choose the **Common Language Runtime Support** for compliance type, such as **Common Language Runtime (old syntax)** for compliance with Managed Extensions for C++ syntax, the CLR programming syntax before Visual Studio 2005.
Use external build system	Specifies to use build tools that aren't included in Visual Studio for building the new project. When this option is selected, you can specify build command lines on the **Specify Debug Configuration Settings** and **Specify Release Configuration Settings** pages.



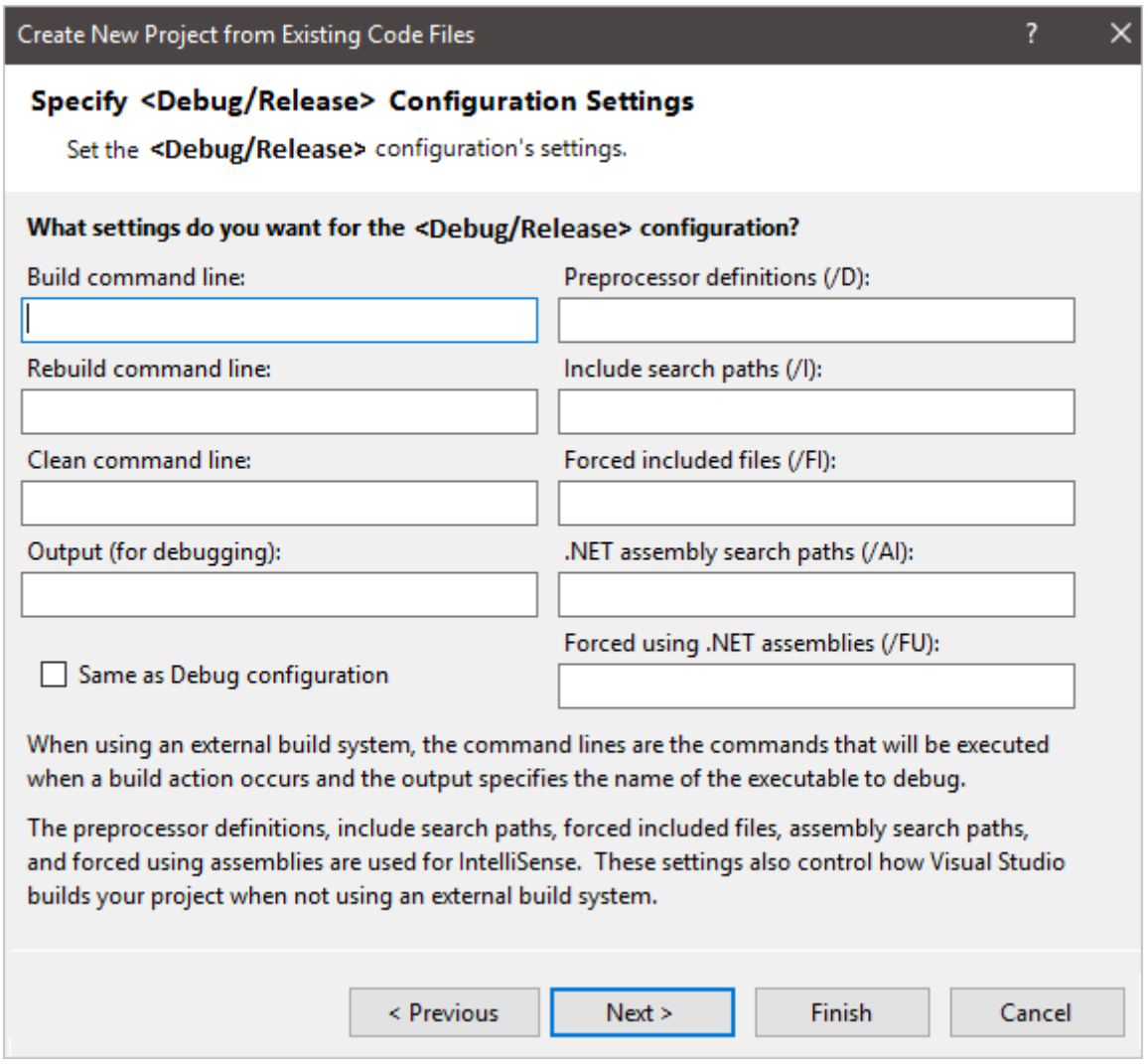
> **Note**

When the **Use External Build System** option is checked, the IDE doesn't build the project, so the /D, /I, /FI, /AI, or /FU options aren't required for compilation. However, these options must be set correctly in order for IntelliSense to function properly.

5. Specify the Debug configuration settings to use. Choose **Next** to continue.

Expand table

Setting	Description
Build command line	Specifies the command line that builds the project. Enter the name of the compiler (plus any switches or arguments) or the build scripts that you want to use to build the project.
Rebuild command line	Specifies the command line that rebuilds the new project.
Clean command line	Specifies the command line to delete support files generated by the build tools for the project.
Output (for debugging)	Specifies the directory path of the output files for the Debug configuration of the project.
Preprocessor definitions (/D)	Defines preprocessor symbols for the project, see /D (Preprocessor Definitions).
Include search path (/I)	Specifies directory paths the compiler searches to resolve file references passed to preprocessor directives in the project, see /I (Additional Include Directories).
Forced included files (/FI)	Specifies header files to process when building the project, see /FI (Name Forced Include File).
.NET assembly search path (/AI)	Specifies the directory paths that the compiler searches to resolve .NET assembly references passed to preprocessor directives in the project, see /AI (Specify Metadata Directories).
Forced using .NET assemblies (/FU)	Specifies .NET assemblies to process when building the project, see /FU (Name Forced #using File).



> ⓘ Note
>
> The **Build**, **Rebuild**, **Clean** command line, and **Output (for debugging)** settings are only enabled if the **Use external build system** option is selected on the **Specify Project Settings** page.

6. Specify the Release configuration settings to use, these settings are the same as the Debug configuration settings.

7. Choose **Finish** to generate the new project.

> ⓘ Note
>
> Here you can check **Same as Debug configuration** to specify that the wizard will generate Release configuration project settings identical to Debug configuration project settings. This option is checked by default. All other options on this page are inactive unless you uncheck this box.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Open Folder support for C++ build systems in Visual Studio

Article • 11/07/2022

In Visual Studio 2017 and later, the "Open Folder" feature enables you to open a folder of source files and immediately start coding with support for IntelliSense, browsing, refactoring, debugging, and so on. As you edit, create, move, or delete files, Visual Studio tracks the changes automatically and continuously updates its IntelliSense index. No .sln or .vcxproj files are loaded; if needed, you can specify custom tasks as well as build and launch parameters through simple .json files. This feature enables you to integrate any third-party build system into Visual Studio. For general information about Open Folder, see Develop code in Visual Studio without projects or solutions.

CMake and Qt

CMake is integrated in the Visual Studio IDE as a component of the C++ desktop workload. The workflow for CMake is not identical to the workflow described in this article. If you are using CMake, see CMake projects in Visual Studio. You can also use CMake to build Qt projects, or you can use the Qt Visual Studio Extension for either Visual Studio 2015 or Visual Studio 2017.

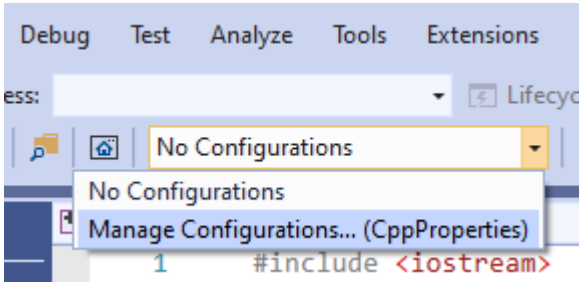
Other build systems

To use the Visual Studio IDE with a build system or compiler toolset that is not directly supported from the main menu select **File | Open | Folder** or press **Ctrl + Shift + Alt + O**. Navigate to the folder that contains your source code files. To build the project, configure IntelliSense and set debugging parameters, you add three JSON files:

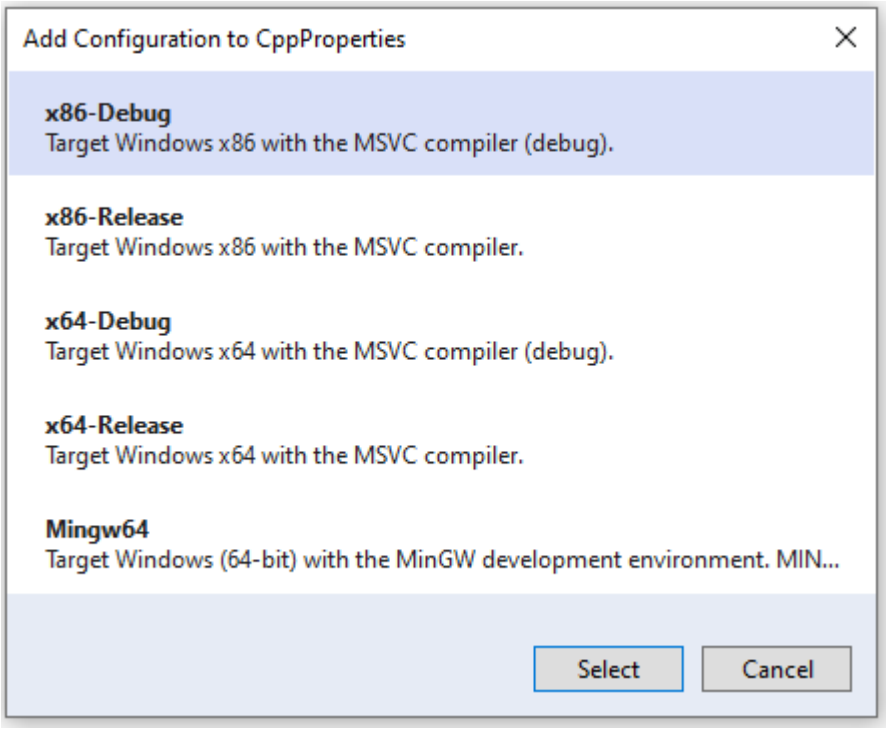
File	Description
CppProperties.json	Specify custom configuration information for browsing. Create this file, if needed, in your root project folder. (Not used in CMake projects.)
tasks.vs.json	Specify custom build commands. Accessed via the **Solution Explorer** context menu item **Configure Tasks**.
launch.vs.json	Specify command line arguments for the debugger. Accessed via the **Solution Explorer** context menu item **Debug and Launch Settings**.

Configure code navigation with CppProperties.json

For IntelliSense and browsing behavior such as **Go to Definition** to work correctly, Visual Studio needs to know which compiler you are using, where the system headers are, and where any additional include files are located if they are not directly in the folder you have opened (the workspace folder). To specify a configuration, you can choose **Manage Configurations** from the dropdown in the main toolbar:



Visual Studio offers the following default configurations:



If, for example, you choose **x64-Debug**, Visual Studio creates a file called *CppProperties.json* in your root project folder:

JSON

```
{
  "configurations": [
    {
      "inheritEnvironments": [
        "msvc_x64"
```

```
      ],
      "name": "x64-Debug",
      "includePath": [
        "${env.INCLUDE}",
        "${workspaceRoot}\\**"
      ],
      "defines": [
        "WIN32",
        "_DEBUG",
        "UNICODE",
        "_UNICODE"
      ],
      "intelliSenseMode": "windows-msvc-x64"
    }
  ]
}
```

This configuration inherits the environment variables of the Visual Studio x64 Developer Command Prompt. One of those variables is `INCLUDE` and you can refer to it here by using the `${env.INCLUDE}` macro. The `includePath` property tells Visual Studio where to look for all the sources that it needs for IntelliSense. In this case, it says "look in the all the directories specified by the INCLUDE environment variable, and also all the directories in the current working folder tree." The `name` property is the name that will appear in the dropdown, and can be anything you like. The `defines` property provides hints to IntelliSense when it encounters conditional compilation blocks. The `intelliSenseMode` property provides some additional hints based on the compiler type. Several options are available for MSVC, GCC, and Clang.

> **Note**
>
> If Visual Studio seems to be ignoring settings in *CppProperties.json*, try adding an exception to your *.gitignore* file like this: `!/CppProperties.json`.

Default configuration for MinGW-w64

If you add the MinGW-W64 configuration, the JSON looks this this:

JSON

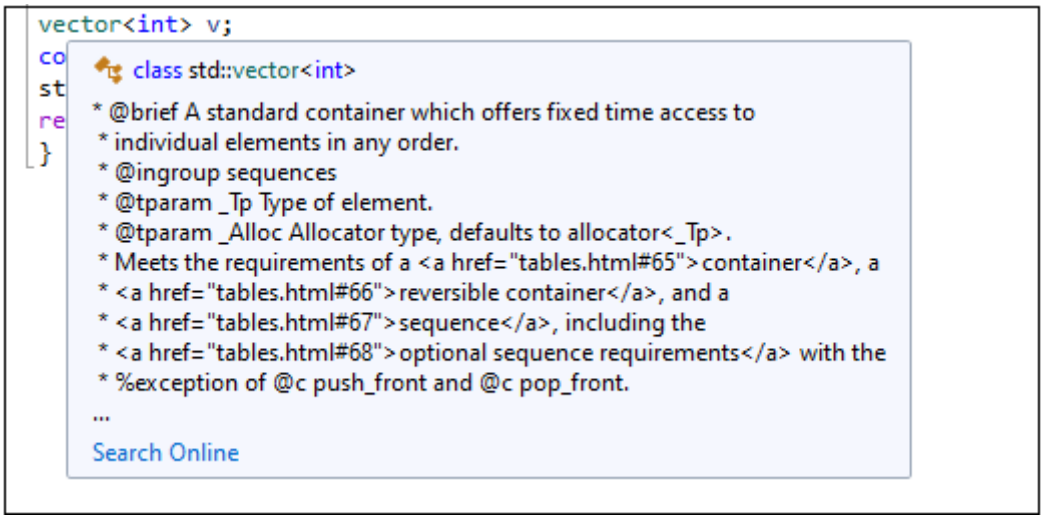
```
{
  "configurations": [
    {
      "inheritEnvironments": [
        "mingw_64"
      ],
```

```
      "name": "Mingw64",
      "includePath": [
        "${env.INCLUDE}",
        "${workspaceRoot}\\**"
      ],
      "intelliSenseMode": "linux-gcc-x64",
      "environments": [
        {
          "MINGW64_ROOT": "C:\\msys64\\mingw64",
          "BIN_ROOT": "${env.MINGW64_ROOT}\\bin",
          "FLAVOR": "x86_64-w64-mingw32",
          "TOOLSET_VERSION": "9.1.0",
          "PATH":
"${env.BIN_ROOT};${env.MINGW64_ROOT}\\..\\usr\\local\\bin;${env.MINGW64_ROOT
}\\..\\usr\\bin;${env.MINGW64_ROOT}\\..\\bin;${env.PATH}",
          "INCLUDE":
"${env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION};${env.MINGW64_ROO
T}\\include\\c++\\${env.TOOLSET_VERSION}\\tr1;${env.MINGW64_ROOT}\\include\\
c++\\${env.TOOLSET_VERSION}\\${env.FLAVOR}",
          "environment": "mingw_64"
        }
      ]
    }
  ]
}
```

Note the `environments` block. It defines properties that behave like environment variables and are available not only in the *CppProperties.json* file, but also in the other configuration files *task.vs.json* and *launch.vs.json*. The `Mingw64` configuration inherits the `mingw_w64` environment, and uses its `INCLUDE` property to specify the value for `includePath`. You can add other paths to this array property as needed.`

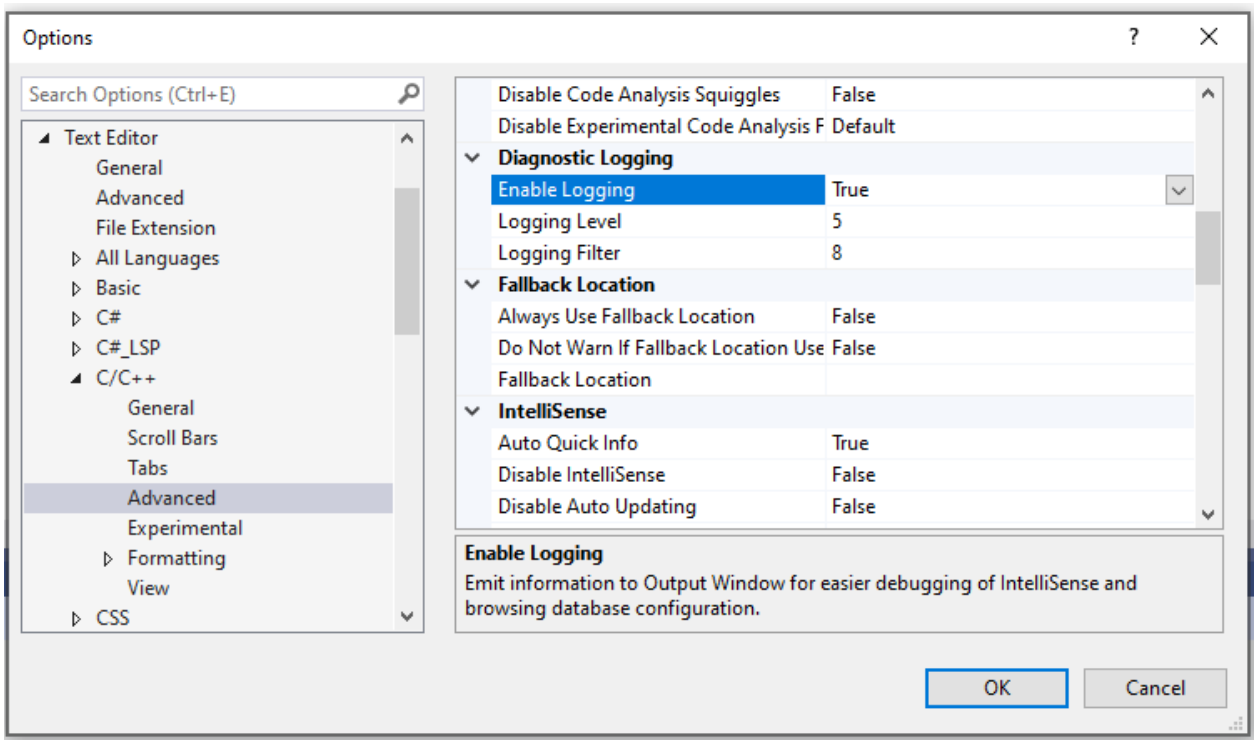
The `intelliSenseMode` property is set to a value appropriate for GCC. For more information on all these properties, see CppProperties schema reference.

When everything is working correctly, you will see IntelliSense from the GCC headers when you hover over a type:



Enable IntelliSense diagnostics

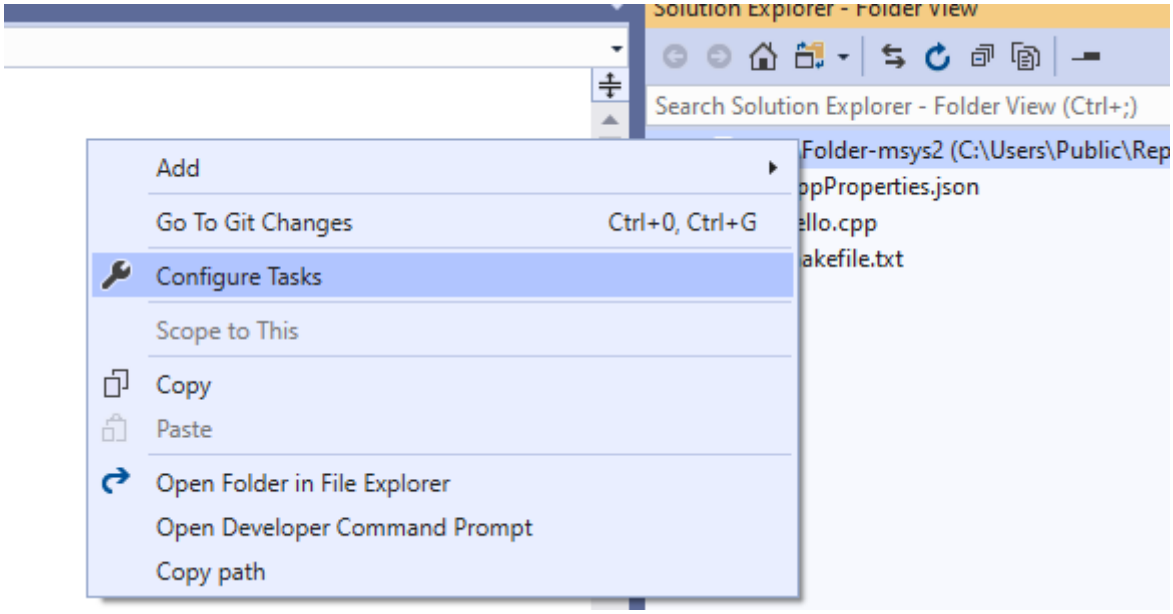
If you are not seeing the IntelliSense that you expect, you can troubleshoot by going to **Tools** > **Options** > **Text Editor** > **C/C++** > **Advanced** and setting **Enable Logging** to `true`. To start with, try setting **Logging Level** to 5, and **Logging Filters** to 8.



Output is piped to the **Output Window** and is visible when you choose **Show Output From: Visual C++ Log*. The output contains, among other things, the list of actual include paths that IntelliSense is trying to use. If the paths do not match the ones in *CppProperties.json*, try closing the folder and deleting the *.vs* subfolder which contains cached browsing data.

Define build tasks with tasks.vs.json

You can automate build scripts or any other external operations on the files you have in your current workspace by running them as tasks directly in the IDE. You can configure a new task by right-clicking on a file or folder and selecting **Configure Tasks**.



This creates (or opens) the *tasks.vs.json* file in the .vs folder which Visual Studio creates in your root project folder. You can define any arbitrary task in this file and then invoke it from the **Solution Explorer** context menu. To continue the GCC example, the following snippet shows a complete *tasks.vs.json* file with as single task that invokes *g++.exe* to build a project. Assume the project contains a single file called *hello.cpp*.

JSON

```
{
  "version": "0.2.1",
  "tasks": [
    {
      "taskLabel": "build hello",
      "appliesTo": "/",
      "type": "default",
      "command": "g++",
      "args": [
        "-g",
        "-o",
        "hello",
        "hello.cpp"
      ]
    }
  ]
}
```

The JSON file is placed in the *.vs* subfolder. To see that folder, click on the **Show All Files** button at the top of **Solution Explorer**. You can run this task by right-clicking on the

root node in **Solution Explorer** and choosing **build hello**. When the task completes you should see a new file, *hello.exe* in **Solution Explorer**.

You can define many kinds of tasks. The following example shows a *tasks.vs.json file* that defines a single task. `taskLabel` defines the name that appears in the context menu. `appliesTo` defines which files the command can be performed on. The `command` property refers to the COMSPEC environment variable, which identifies the path for the console (*cmd.exe* on Windows). You can also reference environment variables that are declared in CppProperties.json or CMakeSettings.json. The `args` property specifies the command line to be invoked. The `${file}` macro retrieves the selected file in **Solution Explorer**. The following example will display the filename of the currently selected .cpp file.

JSON

```
{
  "version": "0.2.1",
  "tasks": [
    {
      "taskLabel": "Echo filename",
      "appliesTo": "*.cpp",
      "type": "command",
      "command": "${env.COMSPEC}",
      "args": ["echo ${file}"]
    }
  ]
}
```

After saving *tasks.vs.json*, you can right-click any *.cpp* file in the folder, choose **Echo filename** from the context menu, and see the file name displayed in the Output window.

For more information, see Tasks.vs.json schema reference.

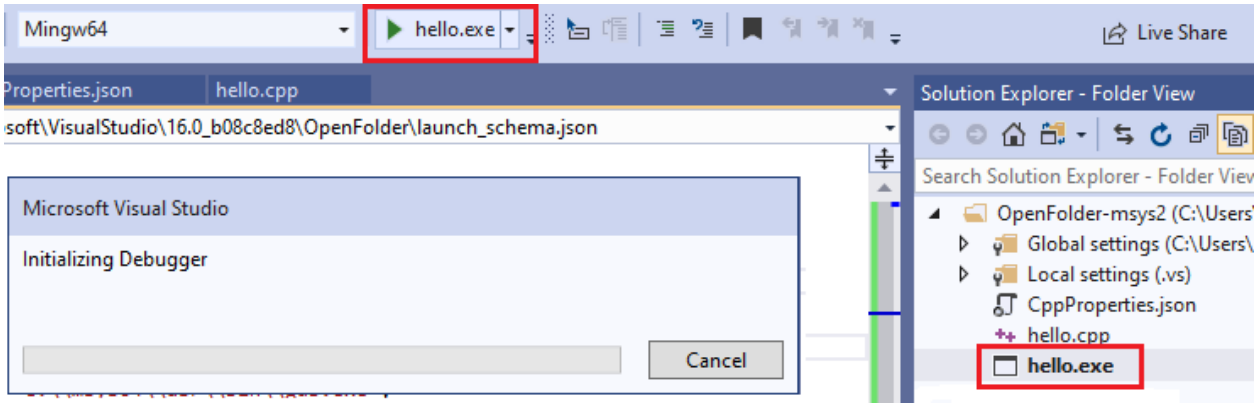
Configure debugging parameters with launch.vs.json

To customize your program's command line arguments and debugging instructions, right-click on the executable in **Solution Explorer** and select **Debug and Launch Settings**. This will open an existing *launch.vs.json* file, or if none exists, it will create a new file with a set of minimal launch settings. First you are given a choice of what kind of debug session you want to configure. For debugging a MinGw-w64 project, we choose **C/C++ Launch for MinGW/Cygwin (gdb)**. This creates a launch configuration for using *gdb.exe* with some educated guesses about default values. One of those default values is `MINGW_PREFIX`. You can substitute the literal path (as shown below) or you can define a `MINGW_PREFIX` property in *CppProperties.json*:

JSON

```
{
  "version": "0.2.1",
  "defaults": {},
  "configurations": [
    {
      "type": "cppdbg",
      "name": "hello.exe",
      "project": "hello.exe",
      "cwd": "${workspaceRoot}",
      "program": "${debugInfo.target}",
      "MIMode": "gdb",
      "miDebuggerPath": "c:\\msys64\\usr\\bin\\gdb.exe",
      "externalConsole": true
    }
  ]
}
```

To start debugging, choose the executable in the debug dropdown, then click the green arrow:



You should see the **Initializing Debugger** dialog and then an external console window that is running your program.

For more information, see launch.vs.json schema reference.

Launching other executables

You can define launch settings for any executable on your computer. The following example launches *7za* and specifies additional arguments, by adding them to the `args` JSON array:

JSON

```
{
  "version": "0.2.1",
```

```
  "defaults": {},
  "configurations": [
    {
      "type": "default",
      "project": "CPP\\7zip\\Bundles\\Alone\\O\\7za.exe",
      "name": "7za.exe list content of helloworld.zip",
      "args": [ "l", "d:\\sources\\helloworld.zip" ]
    }
  ]
}
```

When you save this file, the new configuration appears in the Debug Target dropdown and you can select it to start the debugger. You can create as many debug configurations as you like, for any number of executables. If you press **F5** now, the debugger will launch and hit any breakpoint you may have already set. All the familiar debugger windows and their functionality are now available.

`CppProperties.json` reference

Article • 09/20/2022

Open Folder projects that don't use CMake can store project configuration settings for IntelliSense in a *`CppProperties.json`* file. (CMake projects use a CMakeSettings.json file.) A configuration consists of name/value pairs and defines #include paths, compiler switches, and other parameters. For more information about how to add configurations in an Open Folder project, see Open Folder projects for C++. The following sections summarize the various settings. For a complete description of the schema, navigate to *CppProperties_schema.json*, whose full path is given at the top of the code editor when *`CppProperties.json`* is open.

Configuration properties

A configuration may have any of the following properties:

Name	Description
`inheritEnvironments`	Specifies which environments apply to this configuration.
`name`	The configuration name that will appear in the C++ configuration dropdown
`includePath`	A comma-separated list of folders that should be specified in the include path (maps to `/I` for most compilers)
`defines`	The list of macros that should be defined (maps to `/D` for most compilers)
`compilerSwitches`	One or more additional switches that can influence IntelliSense behavior
`forcedInclude`	Header to be automatically included in every compilation unit (maps to `/FI` for MSVC or `-include` for clang)
`undefines`	The list of macros to be undefined (maps to `/U` for MSVC)
`intelliSenseMode`	The IntelliSense engine to be used. You can specify one of the predefined architecture-specific variants for MSVC, gcc, or Clang.
`environments`	User-defined sets of variables that behave like environment variables in a command prompt and are accessed with the `${env.VARIABLE}` macro.

intelliSenseMode values

The code editor shows the available options when you start to type:



This list shows the supported values:

- `windows-msvc-x86`
- `windows-msvc-x64`
- `windows-msvc-arm`
- `windows-msvc-arm64`
- `android-clang-x86`
- `android-clang-x64`
- `android-clang-arm`
- `android-clang-arm64`
- `ios-clang-x86`
- `ios-clang-x64`
- `ios-clang-arm`
- `ios-clang-arm64`
- `windows-clang-x86`
- `windows-clang-x64`
- `windows-clang-arm`
- `windows-clang-arm64`
- `linux-gcc-x86`
- `linux-gcc-x64`
- `linux-gcc-arm`

Note: The values `msvc-x86` and `msvc-x64` are supported for legacy reasons only. Use the `windows-msvc-*` variants instead.

Pre-defined Environments

Visual Studio provides the following predefined environments for Microsoft C++ which map to the corresponding Developer Command Prompt. When you inherit one of these environments, you can refer to any of the environment variables by using the global property `env` with this macro syntax: `${env.VARIABLE}`.

Variable Name	Description
`vsdev`	The default Visual Studio environment
`msvc_x86`	Compile for x86 using x86 tools
`msvc_x64`	Compile for AMD64 using 64-bit tools
`msvc_arm`	Compile for ARM using x86 tools
`msvc_arm64`	Compile for ARM64 using x86 tools
`msvc_x86_x64`	Compile for AMD64 using x86 tools
`msvc_arm_x64`	Compile for ARM using 64-bit tools
`msvc_arm64_x64`	Compile for ARM64 using 64-bit tools

When the Linux workload is installed, the following environments are available for remotely targeting Linux and WSL:

Variable Name	Description
`linux_x86`	Target x86 Linux remotely
`linux_x64`	Target x64 Linux remotely
`linux_arm`	Target ARM Linux remotely

User-defined environments

You can optionally use the `environments` property to define sets of variables in *`CppProperties.json`* either globally or per-configuration. These variables behave like environment variables in the context of an Open Folder project. You can access them with the `${env.VARIABLE}` syntax from *`tasks.vs.json`* and *`launch.vs.json`* after they're defined here. However, they aren't necessarily set as actual environment variables in any command prompt that Visual Studio uses internally.

Visual Studio 2019 version 16.4 and later: Configuration-specific variables defined in *`CppProperties.json`* are automatically picked up by debug targets and tasks without the

need to set `inheritEnvironments`. Debug targets are launched automatically with the environment you specify in *`CppProperties.json`*.

Visual Studio 2019 version 16.3 and earlier: When you consume an environment, then you have to specify it in the `inheritsEnvironments` property even if the environment is defined as part of the same configuration; the `environment` property specifies the name of the environment. The following example shows a sample configuration for enabling IntelliSense for GCC in an MSYS2 installation. Note how the configuration both defines and inherits the `mingw_64` environment, and how the `includePath` property can access the `INCLUDE` variable.

JSON

```
"configurations": [
    {

      "inheritEnvironments": [
        "mingw_64"
      ],
      "name": "Mingw64",
      "includePath ,": [
        "${env.INCLUDE}",
        "${workspaceRoot}\\**",
      ],
      "intelliSenseMode": "linux-gcc-x64",
      "environments": [
        {
          "MINGW64_ROOT": "C:\\msys64\\mingw64",
          "BIN_ROOT": "${env.MINGW64_ROOT}\\bin",
          "FLAVOR": "x86_64-w64-mingw32",
          "TOOLSET_VERSION": "9.1.0",
          "PATH":
"${env.MINGW64_ROOT}\\bin;${env.MINGW64_ROOT}\\..\\usr\\local\\bin;${env.MIN
GW64_ROOT}\\..\\usr\\bin;${env.MINGW64_ROOT}\\..\\bin;${env.PATH}",
          "INCLUDE":
"${env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION};${env.MINGW64_ROO
T}\\include\\c++\\${env.TOOLSET_VERSION}\\tr1;${env.MINGW64_ROOT}\\include\\
c++\\${env.TOOLSET_VERSION}\\${env.FLAVOR};",
          "environment": "mingw_64"
        }
      ]
    }
  ]
```

When you define an `"environments"` property inside a configuration, it overrides any global variables that have the same names.

Built-in macros

You have access to the following built-in macros inside `CppProperties.json`:

Macro	Description
`${workspaceRoot}`	The full path to the workspace folder
`${projectRoot}`	The full path to the folder where `CppProperties.json` is placed
`${env.vsInstallDir}`	The full path to the folder where the running instance of Visual Studio is installed

Example

If your project has an include folder and also includes `*windows.h*` and other common headers from the Windows SDK, you may want to update your `CppProperties.json` configuration file with the following includes:

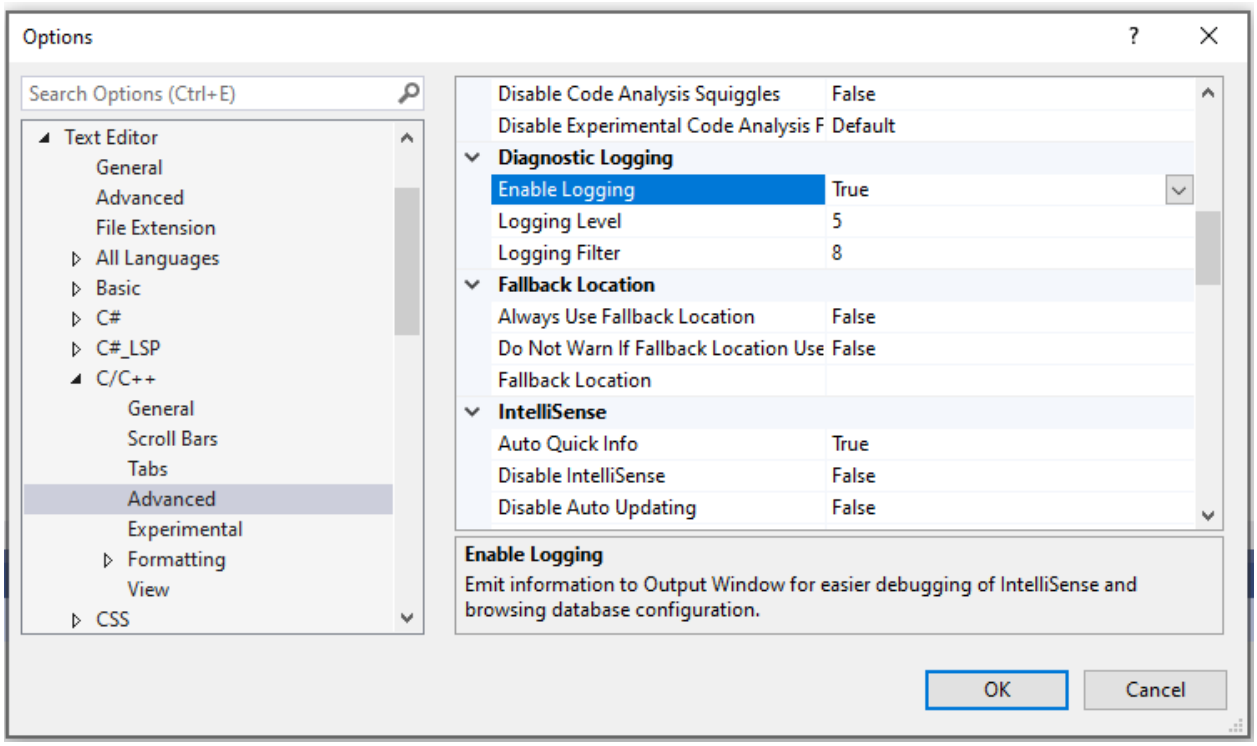
JSON

```
{
  "configurations": [
    {
      "name": "Windows",
      "includePath": [
        // local include folder
        "${workspaceRoot}\\include",
        // Windows SDK and CRT headers
        "${env.WindowsSdkDir}\\include\\${env.WindowsSDKVersion}\\ucrt",
        "${env.NETFXSDKDir}\\include\\um",
        "${env.WindowsSdkDir}\\include\\${env.WindowsSDKVersion}\\um",
        "${env.WindowsSdkDir}\\include\\${env.WindowsSDKVersion}\\shared",
        "${env.VCToolsInstallDir}\\include"
      ]
    }
  ]
}
```

> **Note**
>
> `%WindowsSdkDir%` and `%VCToolsInstallDir%` are not set as global environment variables. Make sure you start `devenv.exe` from a Developer Command Prompt that defines these variables. (Type "developer" in the Windows Start Menu to find a Developer Command Prompt shortcut.)

Troubleshoot IntelliSense errors

If you aren't seeing the IntelliSense that you expect, you can troubleshoot by going to **Tools** > **Options** > **Text Editor** > **C/C++** > **Advanced** and setting **Enable Logging** to `true`. To start with, try setting **Logging Level** to 5, and **Logging Filters** to 8.



Output is piped to the **Output Window** and is visible when you choose **Show Output From: Visual C++ Log**. The output contains, among other things, the list of actual include paths that IntelliSense is trying to use. If the paths don't match the ones in `CppProperties.json`, try closing the folder and deleting the `.vs` subfolder that contains cached browsing data.

To troubleshoot IntelliSense errors caused by missing include paths, open the **Error List** tab, and then filter its output to "IntelliSense only" and error code E1696 "cannot open source file ...".

tasks.vs.json schema reference (C++)

Article • 08/03/2021

To tell Visual Studio how to build your source code in an Open Folder project, add a *tasks.vs.json* file. You can define any arbitrary task here and then invoke it from the **Solution Explorer** context menu. CMake projects do not use this file because all the build commands are specified in *CMakeLists.txt*. For build systems other than CMake, *tasks.vs.json* is where you can specify build commands and invoke build scripts. For general information about using *tasks.vs.json*, see Customize build and debug tasks for "Open Folder" development.

A task has a `type` property which may have one of four values: `default`, `launch`, `remote`, or `msbuild`. Most tasks should use `launch` unless a remote connection is required.

Default Properties

The default properties are available on all types of tasks:

Property	Type	Description
`taskLabel`	string	(Required.) Specifies the task label used in the user interface.
`appliesTo`	string	(Required.) Specifies which files the command can be performed on. The use of wildcards is supported, for example: "", ".cpp", "/*.txt"
`contextType`	string	Allowed values: "custom", "build", "clean", "rebuild". Determines where in the context menu the task will appear. Defaults to "custom".
`output`	string	Specifies an output tag to your task.
`inheritEnvironments`	array	Specifies a set of environment variables inherited from multiple sources. You can define variables in files like *CMakeSettings.json* or *CppProperties.json* and make them available to the task context. **Visual Studio 16.4:**: Specify environment variables on a per-task basis using the `env.VARIABLE_NAME` syntax. To unset a variable, set it to "null".
`passEnvVars`	boolean	Specifies whether or not to include additional environment variables to the task context. These variables are different from the ones defined using the `envVars` property. Defaults to "true".

Launch properties

When the task type is `launch`, these properties are available:

Property	Type	Description
`command`	string	Specifies the full path of the process or script to launch.
`args`	array	Specifies a comma-separated list of arguments passed to the command.
`launchOption`	string	Allowed values: "None", "ContinueOnError","IgnoreError". Specifies how to proceed with the command when there are errors.
`workingDirectory`	string	Specifies the directory in which the command will run. Defaults to the project's current working directory.
`customLaunchCommand`	string	Specifies a global scope customization to apply before executing the command. Useful for setting environment variables like %PATH%.
`customLaunchCommandArgs`	string	Specifies arguments to customLaunchCommand. (Requires `customLaunchCommand`.)
`env`	Specifies a key-value list of custom environment variables. For example, "myEnv": "myVal"	
`commands`	array	Specifies a list of commands to invoke in order.

Example

The following tasks invoke *make.exe* when a makefile is provided in the folder and the `Mingw64` environment has been defined in *CppProperties.json*, as shown in CppProperties.json schema reference:

JSON

```
{
 "version": "0.2.1",
```

```
  "tasks": [
    {
      "taskLabel": "gcc make",
      "appliesTo": "*.cpp",
      "type": "launch",
      "contextType": "custom",
      "inheritEnvironments": [
        "Mingw64"
      ],
      "command": "make"
    },
    {
      "taskLabel": "gcc clean",
      "appliesTo": "*.cpp",
      "type": "launch",
      "contextType": "custom",
      "inheritEnvironments": [
        "Mingw64"
      ],
      "command": "make",
      "args": ["clean"]
    }
  ]
}
```

These tasks can be invoked from the context menu when you right click on a *.cpp* file in **Solution Explorer**.

Remote Properties

Remote tasks are enabled when you install the Linux development with C++ workload and add a connection to a remote machine by using the Visual Studio Connection Manager. A remote task runs commands on a remote system and can also copy files to it.

When the task type is `remote`, these properties are available:

Property	Type	Description
`remoteMachineName`	string	The name of the remote machine. Must match a machine name in **Connection Manager**.
`command`	string	The command to send to the remote machine. By default commands are executed in the $HOME directory on the remote system.
`remoteWorkingDirectory`	string	The current working directory on the remote machine.

Property	Type	Description
`localCopyDirectory`	string	The local directory to copy to the remote machine. Defaults to the current working directory.
`remoteCopyDirectory`	string	The directory on the remote machine into which `localCopyDirectory` is copied.
`remoteCopyMethod`	string	The method to use for copying. Allowed values: "none", "sftp", "rsync". rsync is recommended for large projects.
`remoteCopySourcesOutputVerbosity`	string	Allowed values: "Normal","Verbose","Diagnostic".
`rsyncCommandArgs`	string	Defaults to "-t --delete".
`remoteCopyExclusionList`	array	Comma-separated list of files in `localCopyDirectory` to exclude from copy operations.

Example

The following task will appear in the context menu when you right-click on *main.cpp* in **Solution Explorer**. It depends on a remote machine called `ubuntu` in **Connection Manager**. The task copies the current open folder in Visual Studio into the `sample` directory on the remote machine and then invokes g++ to build the program.

JSON

```
{
  "version": "0.2.1",
  "tasks": [
    {
      "taskLabel": "Build",
      "appliesTo": "main.cpp",
      "type": "remote",
      "contextType": "build",
      "command": "g++ main.cpp",
      "remoteMachineName": "ubuntu",
      "remoteCopyDirectory": "~/sample",
      "remoteCopyMethod": "sftp",
      "remoteWorkingDirectory": "~/sample/hello",
      "remoteCopySourcesOutputVerbosity": "Verbose"
    }
  ]
}
```

MSBuild properties

When the task type is `msbuild`, these properties are available:

Property	Type	Description
`verbosity`	string	Specifies the MSBuild project build output verbosityAllowed values: "Quiet", "Minimal", "Normal", "Detailed", "Diagnostic".
`toolsVersion`	string	Specifies the toolset version to build the project, for example "2.0", "3.5", "4.0", "Current". Defaults to "Current".
`globalProperties`	object	Specifies a key-value list of the global properties to pass into the project, for example, "Configuration":"Release"
`properties`	object	Specifies a key-value list of additional project only properties.
`targets`	array	Specifies the list of targets to invoke, in order, on the project. The project's default target is used if none are specified.

`launch.vs.json` schema reference (C++)

Article • 03/02/2022

In Visual Studio 2017 and later versions, you can open and build code from nearly any directory-based project without requiring a solution or project file. When there's no project or solution file, you can specify custom build tasks and launch parameters through JSON configuration files. This article describes the `launch.vs.json` file, which specifies debugging parameters. For more information about the "Open Folder" feature, see Develop code in Visual Studio without projects or solutions.

To create the file, right-click on an executable file in **Solution Explorer** and choose **Debug and Launch Settings**. Choose the option that most closely matches your project and then use the following properties to modify the configuration as needed. For more information on debugging CMake projects, see Configure CMake debugging sessions.

Default properties

Property	Type	Description
`args`	array	Specifies the command-line arguments passed to the launched program.
`buildConfigurations`	array	A key-value pair that specifies the name of the build mode to apply the configurations. For example, `Debug` or `Release` and the configurations to use according to the selected build mode.
`currentDir`	string	Specifies the full directory path to the Build Target. The directory is detected automatically unless this parameter is set.
`cwd`	string	Full path to the directory on the remote system where the program will run. Defaults to `"${debugInfo.defaultWorkingDirectory}"`
`debugType`	string	Specifies the debugging mode according to the type of code (native, managed, or mixed). The mode is automatically detected unless this parameter is set. Allowed values: `"native"`, `"managed"`, `"mixed"`.
`env`	array	Specifies a key-value list of custom environment variables. For example: `env: {"myEnv":"myVal"}`.
`inheritEnvironments`	array	Specifies a set of environment variables inherited from multiple sources. You can define some variables in files like *`CMakeSettings.json`* or *`CppProperties.json`* and make them available to debug context. **Visual Studio 16.4:** Specify environment variables on a per-target basis using the `env.VARIABLE_NAME` syntax. To unset a variable, set it to `"null"`.
`name`	string	Specifies the name of the entry in the **Startup Item** dropdown.
`noDebug`	boolean	Specifies whether to debug the launched program. The default value for this parameter is **`false`** if not specified.
`portName`	string	Specifies the name of port when attaching to a running process.
`program`	string	The debug command to execute. Defaults to `"${debugInfo.fullTargetPath}"`.
`project`	string	Specifies the relative path to the project file. Normally, you don't need to change this value when debugging a CMake project.

Property	Type	Description
`projectTarget`	string	Specifies the optional target invoked when building `project`. The target must match the name in the **Startup Item** dropdown.
`stopOnEntry`	boolean	Specifies whether to break a soon as the process is launched and the debugger attaches. The default value for this parameter is **`false`**.
`remoteMachine`	string	Specifies the name of the remote machine where the program is launched.
`type`	string	Specifies whether the project is a `dll` or `exe` Defaults to .exe

C++ Linux properties

Property	Type	Description
`program`	string	Full path to program executable on the remote machine. When using CMake, the macro `${debugInfo.fullTargetPath}` can be used as the value of this field.
`processId`	integer	Optional process ID to attach the debugger to.
`sourceFileMap`	object	Optional source file mappings passed to the debug engine. Format: `{ "\<Compiler source location>": "\<Editor source location>" }` or `{ "\<Compiler source location>": { "editorPath": "\<Editor source location>", "useForBreakpoints": true } }`. Example: `{ "/home/user/foo": "C:\\foo" }` or `{ "/home/user/foo": { "editorPath": "c:\\foo", "useForBreakpoints": true } }`. For more information, see Source file map options.
`additionalProperties`	string	One of the sourceFileMapOptions. (See below.)
`MIMode`	string	Indicates the type of MI-enabled console debugger that the MIDebugEngine will connect to. Allowed values are `"gdb"`, `"lldb"`.
`args`	array	Command-line arguments passed to the program.
`environment`	array	Environment variables to add to the environment for the program. Example: `[ { "name": "squid", "value": "clam" } ]`.
`targetArchitecture`	string	The architecture of the debuggee. The architecture is detected automatically unless this parameter is set. Allowed values are `x86`, `arm`, `arm64`, `mips`, `x64`, `amd64`, `x86_64`.
`visualizerFile`	string	The .natvis file to be used when debugging this process. This option isn't compatible with GDB pretty printing. See `"showDisplayString"` if you use this setting.
`showDisplayString`	boolean	When a visualizerFile is specified, `showDisplayString` will enable the display string. Turning on this option can slow performance during debugging.
`remoteMachineName`	string	The remote Linux machine that hosts gdb and the program to debug. Use the Connection Manager for adding new Linux machines. When using CMake, the macro `${debugInfo.remoteMachineName}` can be used as the value of this field.
`miDebuggerPath`	string	The path to the MI-enabled debugger (such as gdb). When unspecified, it will search PATH first for the debugger.
`miDebuggerServerAddress`	string	Network address of the MI-enabled debugger server to connect to. Example: `"localhost:1234"`.

Property	Type	Description
`setupCommands`	array	One or more GDB/LLDB commands to execute to set up the underlying debugger. Example: `"setupCommands": [ { "text": "-enable-pretty-printing", "description": "Enable GDB pretty printing", "ignoreFailures": true }]`. For more information, see Launch setup commands.
`customLaunchSetupCommands`	array	If provided, this value replaces the default commands used to launch a target with some other commands. For example, use "-target-attach" to attach to a target process. An empty command list replaces the launch commands with nothing, which can be useful if the debugger is being provided launch options as command-line options. Example: `"customLaunchSetupCommands": [ { "text": "target-run", "description": "run target", "ignoreFailures": false }]`.
`launchCompleteCommand`	string	The command to execute after the debugger is fully set up, to cause the target process to run. Allowed values are "exec-run", "exec-continue", "None". The default value is "exec-run".
`debugServerPath`	string	Optional full path to debug server to launch. Defaults to null.
`debugServerArgs`	string	Optional debug server args. Defaults to null.
`filterStderr`	boolean	Search stderr stream for server-started pattern and log stderr to debug output. Defaults to **`false`**.
`coreDumpPath`	string	Optional full path to a core dump file for the specified program. Defaults to null.
externalConsole	boolean	If true, a console is launched for the debuggee. If **`false`**, no console is launched. The default for this setting is **`false`**. This option is ignored in some cases for technical reasons.
`pipeTransport`	string	When present, this value tells the debugger to connect to a remote computer using another executable as a pipe that will relay standard input/output between Visual Studio and the MI-enabled debugger (such as gdb). Allowed values: one or more Pipe Transport Options.

debugInfo macros

The following macros provide information about the debugging environment. They're useful for customizing the launch of your app for debugging.

Macro	Description	Example
`addressSanitizerRuntimeFlags`	Runtime flags used to customize behavior of the address sanitizer. Used to set the environment variable `"ASAN_OPTIONS"`.	`"env": {"ASAN_OPTIONS": "${addressSanitizerRuntimeFlags}:anotherFlag=true"}`
`defaultWorkingDirectory`	Set to the directory part of `"fullTargetPath"`. If the CMake variable `VS_DEBUGGER_WORKING_DIRECTORY` is defined, then `defaultWorkingDirectory` is set to that value, instead.	`"cwd":"${debugInfo.defaultWorkingDirectory}"`

Macro	Description	Example
`fullTargetPath`	The full path to the binary being debugged.	`"program": "${debugInfo.fullTargetPath}"`
`linuxNatvisPath`	The full windows path to the VS linux `.natvis` file. Usually appears as the value `"visualizerFile"`.	
`parentProcessId`	The process ID for the current Visual Studio instance. Used as a parameter to shellexec.	See pipeTransport example below.
`remoteMachineId`	A unique, numeric identifier for the connection to the remote machine. Used as a parameter to shellexec.	See pipeTransport example below.
`remoteWorkspaceRoot`	Linux path to the remote copy of the workspace.	Specify file locations on the remote machine. For example: `"args": ["${debugInfo.remoteWorkspaceRoot}/Data/MyInputFile.dat"]`
`resolvedRemoteMachineName`	The name of the target remote machine.	`"targetMachine"` value in a deployment directive
`shellexecPath`	The path to the shellexec program that Visual Studio is using to manage the remote machine connection.	See pipeTransport example below
`tty`	gdb will redirect input and output to this device for the program being debugged. Used as a parameter to gdb (-tty).	See pipeTransport example below.
`windowsSubsystemPath`	The full path to the Windows Subsystem for Linux instance.	

The pipeTransport example below shows how to use some of the `debugInfo` macros defined above:

JSON

```
"pipeTransport": {
    "pipeProgram": "${debugInfo.shellexecPath}",
    "pipeArgs": [
        "/s",
        "${debugInfo.remoteMachineId}",
        "/p",
        "${debugInfo.parentProcessId}",
        "/c",
        "${debuggerCommand}",
        "--tty=${debugInfo.tty}"
    ],
    "pipeCmd": [
        "/s",
        "${debugInfo.remoteMachineId}",
        "/p",
        "${debugInfo.parentProcessId}",
        "/c",
```

```
        "${debuggerCommand}"
    ]
    }
```

C++ Windows remote debug and deploy properties

Used when debugging and deploying an app on a remote machine.

Property	Type	Description
`cwd`	string	The working directory of the target on the remote machine. When using CMake, the macro `${debugInfo.defaultWorkingDirectory}` can be used as the value of this field. The default value is the directory of the debug program/command.
`deploy`	string	Specifies extra files or directories to deploy. For example: `"deploy": {"sourcePath":"<Full path to source file/directory on host machine>", "targetPath":"<Full destination path to file/directory on target machine>"}`
`deployDirectory`	string	The location on the remote machine where project outputs are automatically deployed to. Defaults to "`C:\Windows Default Deploy Directory\<name of app>`
`deployDebugRuntimeLibraries`	string	Specifies whether to deploy the debug runtime libraries for the active platform. Defaults to `"true"` if the active configurationType is `"Debug"`
`deployRuntimeLibraries`	string	Specifies whether to deploy the runtime libraries for the active platform. Defaults to `"true"` if the active configurationType is `"MinSizeRel"`, `"RelWithDebInfo"`, or `"Release"`.
`disableDeploy`	boolean	Specifies whether files should be deployed.
`remoteMachineName`	string	Specifies the name of the remote ARM64 Windows machine where the program is launched. May be the server name or the remote machine's IP address.
`authenticationType`	string	Specifies the type of remote connection. Possible values are `"windows"` and `"none"`. The default is `"windows"`. This value should match the authentication setting specified on the remote debugger that is running on the remote machine.

Launch setup commands

Used with the `setupCommands` property:

Property	Type	Description
`text`	string	The debugger command to execute.
`description`	string	Optional description for the command.
`ignoreFailures`	boolean	If true, failures from the command should be ignored. Defaults to `false`.

Pipe transport options

Used with the `pipeTransport` property:

Property	Type	Description
`pipeCwd`	string	The fully qualified path to the working directory for the pipe program.
`pipeProgram`	string	The fully qualified pipe command to execute.
`pipeArgs`	array	Command-line arguments passed to the pipe program to configure the connection.
`debuggerPath`	string	The full path to the debugger on the target machine, for example /usr/bin/gdb.
`pipeEnv`	object	Environment variables passed to the pipe program.
`quoteArgs`	boolean	If individual arguments contain characters (such as spaces or tabs), should it be quoted? If **`false`**, the debugger command will no longer be automatically quoted. Default is **`true`**.

Source file map options

Use with the `sourceFileMap` property:

Property	Type	Description
`editorPath`	string	The location of the source code for the editor to locate.
`useForBreakpoints`	boolean	When setting breakpoints, this source mapping should be used. If **`false`**, only the filename and line number is used for setting breakpoints. If **`true`**, breakpoints will be set with the full path to the file and line number only when this source mapping is used. Otherwise just filename and line number will be used when setting breakpoints. Default is **`true`**.

CMake projects in Visual Studio

Article • 03/20/2024

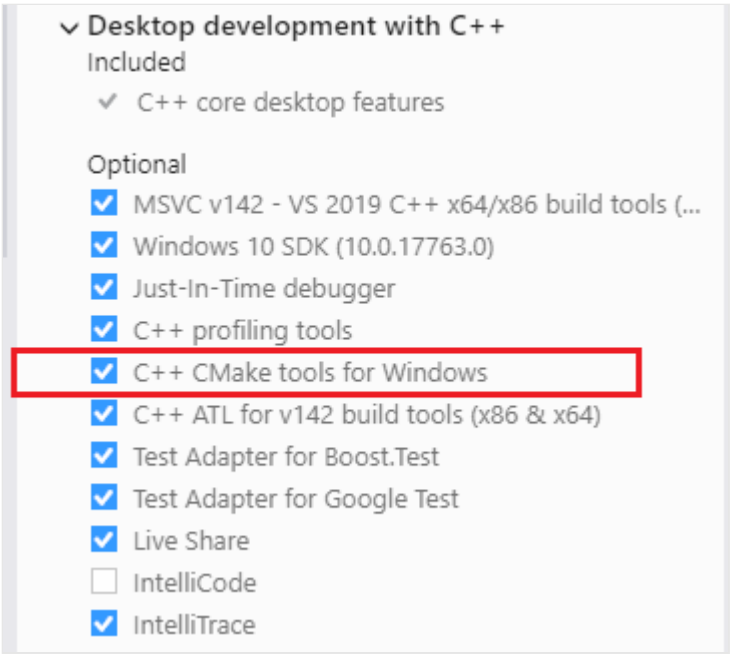
CMake is a cross-platform, open-source tool for defining build processes that run on multiple platforms. This article assumes you're familiar with CMake. For more information about CMake, see the CMake documentation. The CMake tutorial is a good starting point to learn more.

> **Note**
>
> CMake has become more and more integrated with Visual Studio over the past few releases. To see the documentation for your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

Visual Studio's native support for CMake enables you to edit, build, and debug CMake projects on Windows, the Windows Subsystem for Linux (WSL), and remote systems from the same instance of Visual Studio. CMake project files (such as `CMakeLists.txt`) are consumed directly by Visual Studio for the purposes of IntelliSense and browsing. `cmake.exe` is invoked directly by Visual Studio for CMake configuration and build.

Installation

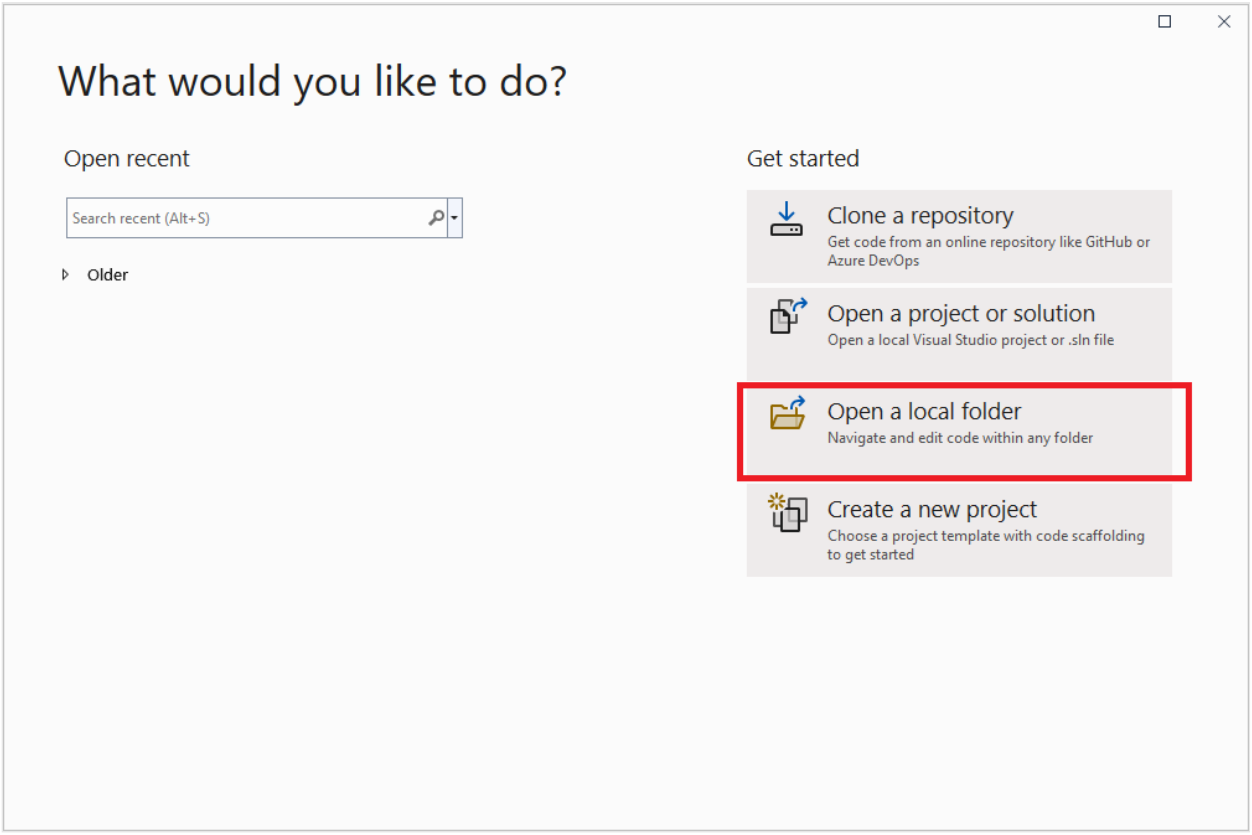
C++ CMake tools for Windows is installed as part of the **Desktop development with C++** and **Linux Development with C++** workloads. Both **C++ CMake tools for Windows** and **Linux Development with C++** are required for cross-platform CMake development.



For more information, see Install the C++ Linux workload in Visual Studio.

IDE integration

When you **open a folder** containing a `CMakeLists.txt` file, the following things happen.

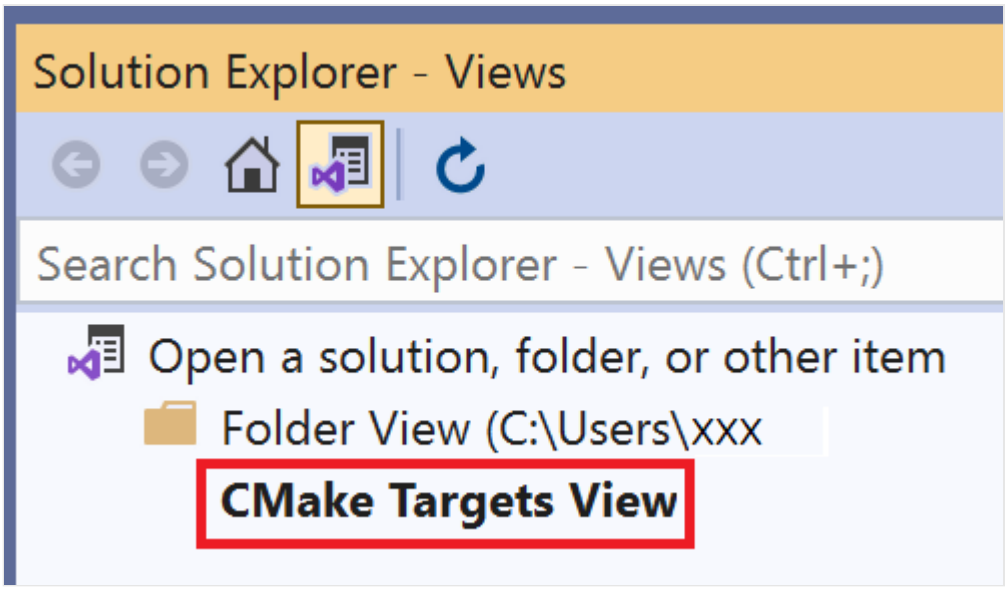


- Visual Studio adds **CMake** items to the **Project** menu, with commands for viewing and editing CMake scripts.
- The **Solution Explorer** displays the folder structure and files.

- Visual Studio runs CMake and generates the CMake cache file (`CMakeCache.txt`) for the default configuration. The CMake command line is displayed in the **Output Window**, along with other output from CMake.
- In the background, Visual Studio starts to index the source files to enable IntelliSense, browsing information, refactoring, and so on. As you work, Visual Studio monitors changes in the editor and also on disk to keep its index in sync with the sources.

> ⓘ **Note**
>
> Starting in Visual Studio 2022 version 17.1 Preview 2, if your top-level `CMakeLists.txt` exists in a subfolder and not at the root of the workspace, you'll be prompted whether you'd like to enable CMake integration or not. For more information, see CMake partial activation.

Once CMake cache generation has succeeded, you can also view your projects organized logically by targets. Choose the **Select View** button on the **Solution Explorer** toolbar. From the list in **Solution Explorer - Views**, select **CMake Targets View** and press **Enter** to open the targets view:



Choose the **Show All Files** button at the top of **Solution Explorer** to see all the CMake-generated output in the `out/build/<config>` folders.

Use the `CMakeLists.txt` file in each project folder just as you would in any CMake project. You can specify source files, find libraries, set compiler and linker options, and specify other build system-related information. For more information on CMake language services provided by Visual Studio, see Editing CMakeLists.txt files.

Visual Studio uses a CMake configuration file to drive CMake cache generation and build. For more information, see Configuring CMake projects and Building CMake projects.

To pass arguments to an executable at debug time, you can use another file called `launch.vs.json`. For more information on debugging cross-platform CMake projects in Visual Studio, see Debugging CMake projects.

Most Visual Studio and C++ language features are supported by CMake projects in Visual Studio. Examples include:

- Edit and Continue for CMake projects
- Incredibuild integration for CMake projects
- AddressSanitizer support for CMake projects
- Clang/LLVM support

> **Note**
>
> For other kinds of Open Folder projects, an additional JSON file `CppProperties.json` is used. This file is not relevant for CMake projects.

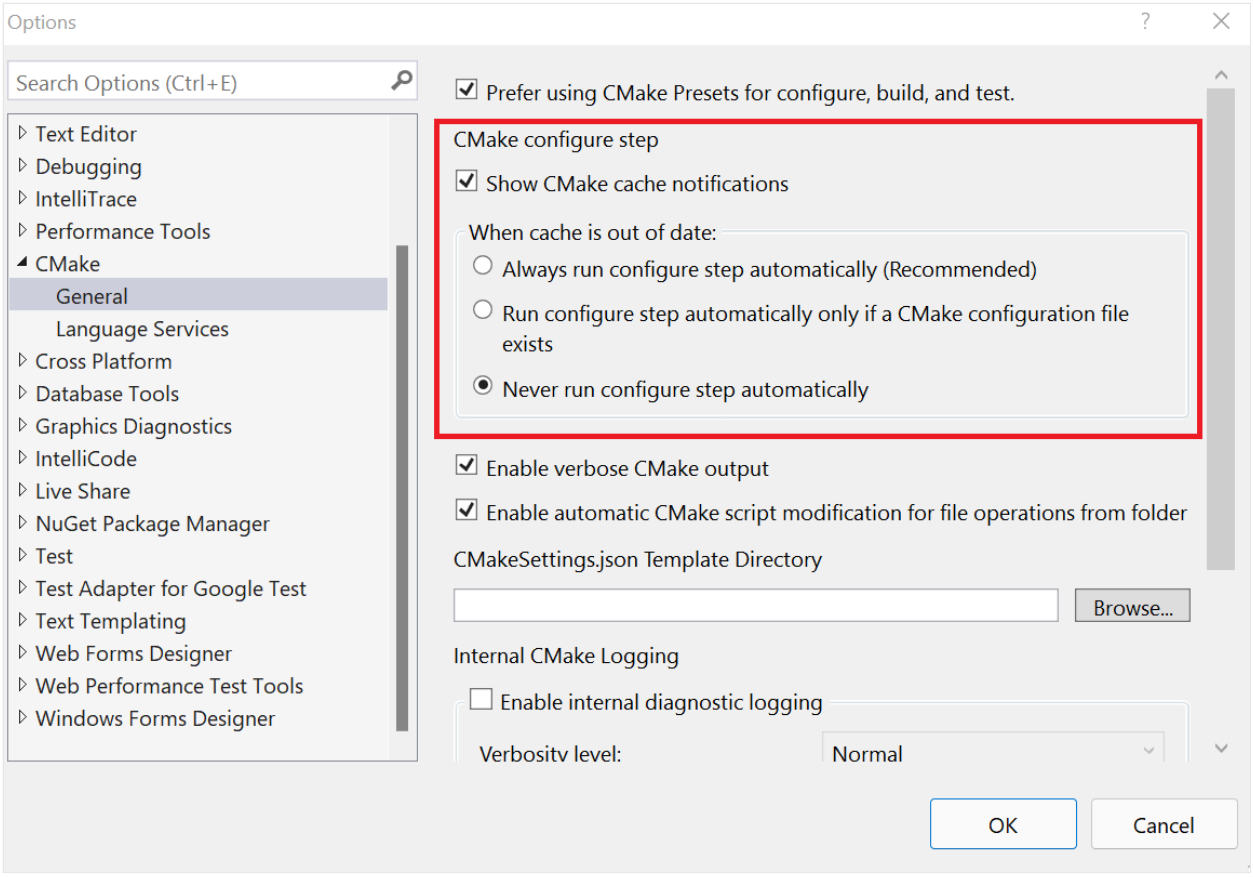
Configuring CMake projects

The CMake configure step generates the project build system. It's equivalent to invoking `cmake.exe` from the command line. For more information on the CMake configure step, see the CMake documentation.

Visual Studio uses a CMake configuration file to drive CMake generation and build. `CMakePresets.json` is supported by Visual Studio 2019 version 16.10 or later and is the recommended CMake configuration file. `CMakePresets.json` is supported directly by CMake and can be used to drive CMake generation and build from Visual Studio, from VS Code, in a Continuous Integration pipeline, and from the command line on Windows, Linux, and Mac. For more information on `CMakePresets.json`, see Configure and build with CMake Presets. `CMakeSettings.json` is available for customers using an earlier version of Visual Studio. For more information on `CMakeSettings.json`, see Customize CMake build settings.

When you make significant changes to your CMake configuration file or a `CMakeLists.txt` file, Visual Studio will automatically run the CMake configure step. You can invoke the configure step manually: Select **Project** > **Configure Cache** from the

toolbar. You can also change your configuration preferences in **Tools** > **Options** > **CMake** > **General**.



If the configure step finishes without errors, then the information that's available drives C++ IntelliSense and language services. It's also used in build and debug operations.

You can also open an existing CMake cache in Visual Studio. For more information, see Open an existing cache.

Customize configuration feedback and notifications

By default, most configuration messages are suppressed unless there's an error. To see all messages, select **Tools** > **Options** > **CMake** > **Enable verbose CMake diagnostic output**.

You can also disable all CMake cache notifications (gold bars) by deselecting **Show CMake cache notification**.

Troubleshooting CMake cache errors

If you need more information about the state of the CMake cache to diagnose a problem, open the **Project** main menu or the `CMakeLists.txt` context menu in **Solution Explorer** to run one of these commands:

- **View CMakeCache.txt** opens the `CMakeCache.txt` file from the build directory in the editor. Any edits you make here to `CMakeCache.txt` are wiped out if you clean the cache. To make changes that persist after you clean the cache, see Customize CMake settings or Configure and build with CMake Presets.
- **Delete Cache and Reconfigure** deletes the build directory and reconfigures from a clean cache.
- **Configure Cache** forces the generate step to run even if Visual Studio considers the environment up to date.

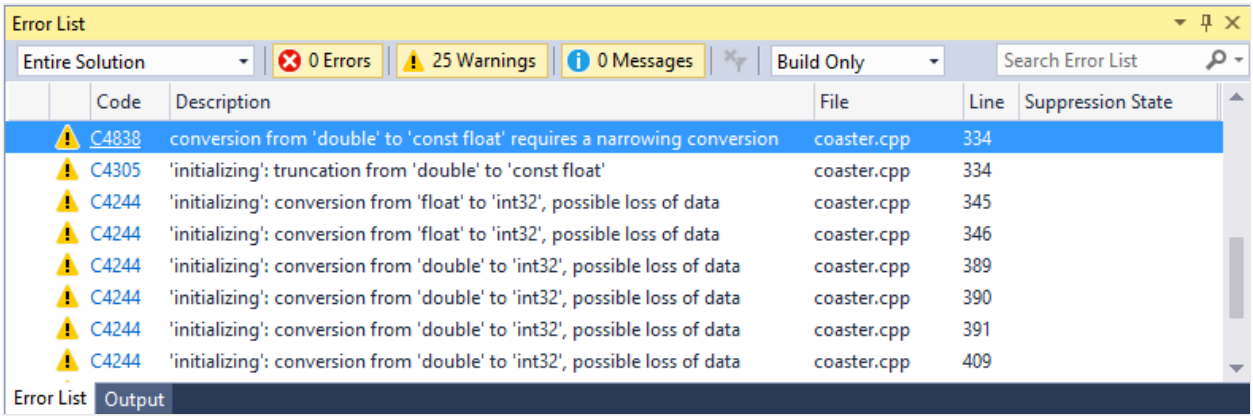
Building CMake projects

The CMake build step builds an already generated project binary tree. It's equivalent to invoking `cmake --build` from the command line. For more information on the CMake build step, see the CMake documentation.

To build a CMake project, you have these choices:

1. In the toolbar, find the **Startup Item** dropdown. Select the preferred target and press **F5**, or choose the **Run** button on the toolbar. The project automatically builds first, just like a Visual Studio solution.
2. Right-click on CMake target with **CMake Targets View** active in the **Solution Explorer** and select **Build** from the context menu.
3. From the main menu, select **Build** > **Build All**. Make sure that a CMake target is already selected in the **Startup Item** dropdown in the toolbar.

As you would expect, build results are shown in the **Output Window** and **Error List**.



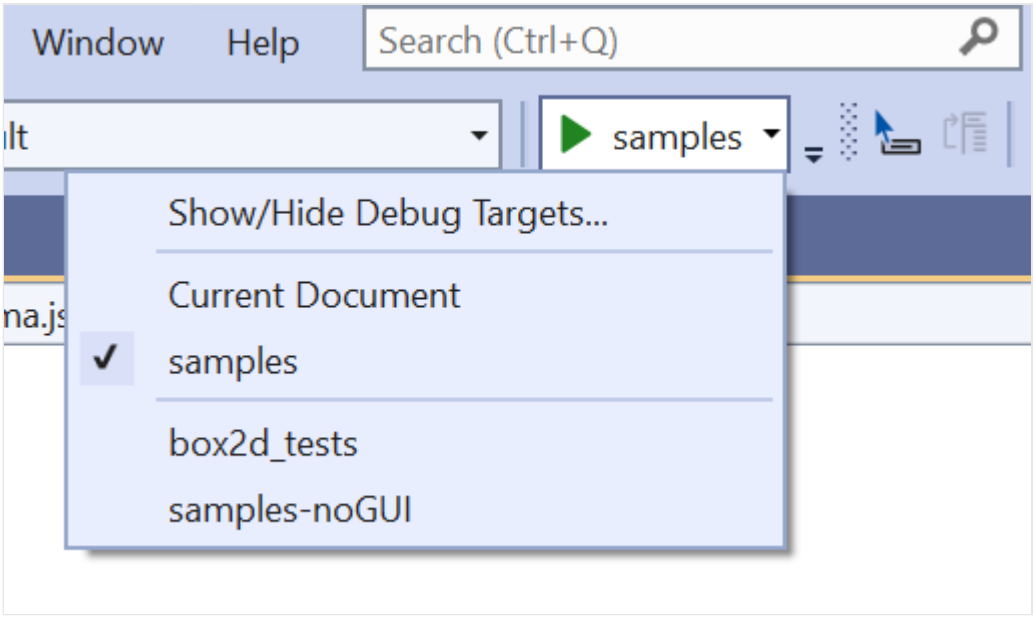
CMake build warnings about conversions that may result in data loss such as converting from a float to an integer, are visible. :::image-end:::

Edit build settings

Visual Studio uses a CMake configuration file to drive CMake builds. CMake configuration files encapsulate build options like native build tool switches and environment variables. If `CMakePresets.json` is your active configuration file, see Configure and build with CMake Presets. If `CMakeSettings.json` is your active configuration file, see Customize CMake build settings. `CMakePresets.json` is available in Visual Studio 2019 version 16.10 or later and is the recommended CMake configuration file.

Debugging CMake projects

All executable CMake targets are shown in the **Startup Item** dropdown in the toolbar. To start debugging, select one and press the **Debug** > **Start Debugging** button in the toolbar. In a CMake project, the "Current document" option is only valid for .cpp files.



The **Debug** or **F5** commands first build the project if changes have been made since the previous build. Changes to the CMake configuration file (`CMakePresets.json` or `CMakeSettings.json`) or a `CMakeLists.txt` causes the CMake cache to be regenerated.

You can customize a CMake debugging session by setting properties in the `launch.vs.json` file. To customize debug settings for a specific target, select the target in the **Startup Item** dropdown and press **Debug** > **Debug and Launch Settings for <active-target>**. For more information on CMake debugging sessions, see Configure CMake debugging sessions.

Just My Code for CMake projects

When you build for Windows using the MSVC compiler, CMake projects have support for Just My Code debugging. To change the Just My Code setting, go to **Tools** > **Options** > **Debugging** > **General**.

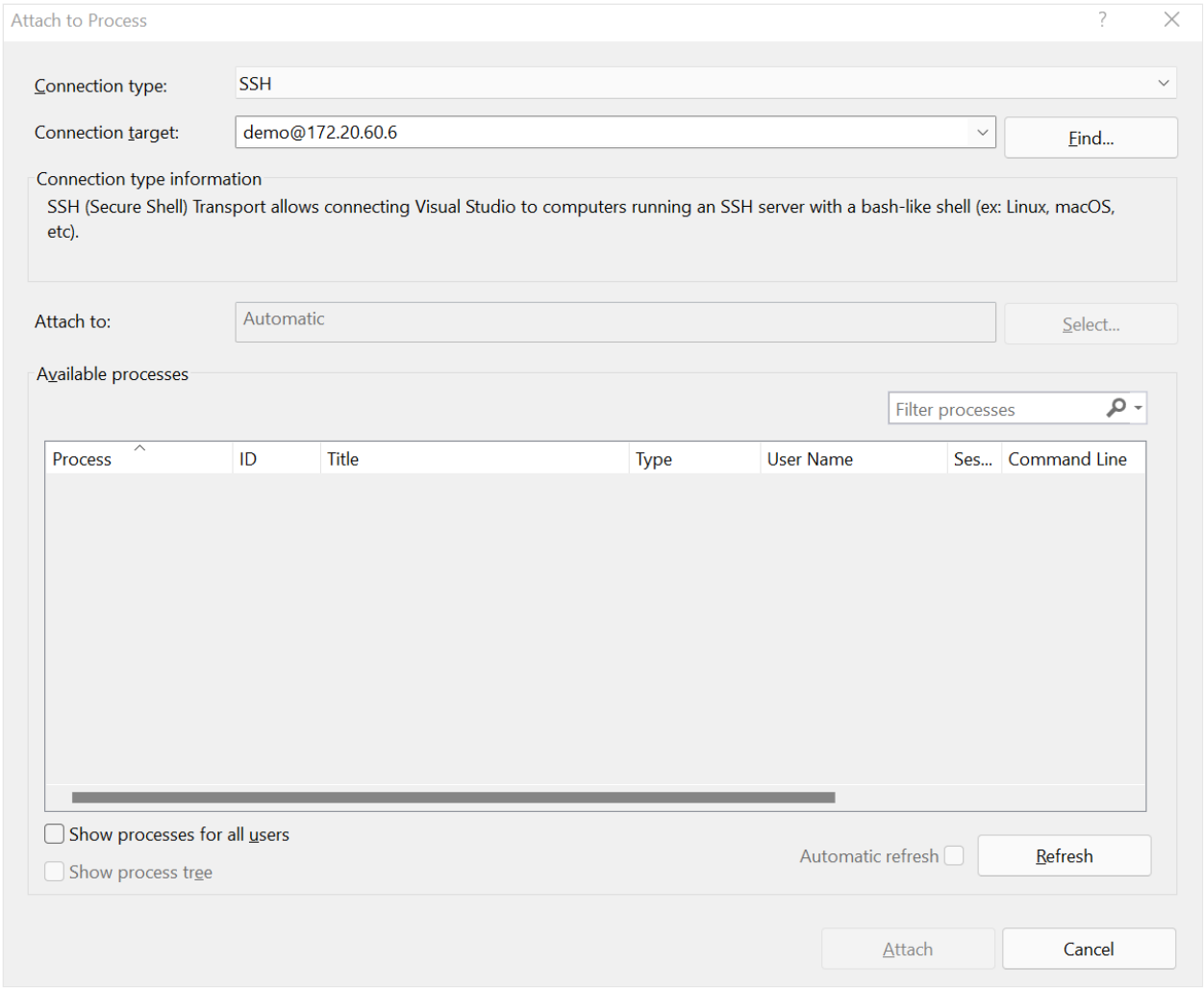
Edit and Continue for CMake projects

When you build for Windows with the MSVC compiler, CMake projects have support for Edit and Continue. Add the following code to your `CMakeLists.txt` file to enable Edit and Continue.

```
if(MSVC)
  target_compile_options(<target> PUBLIC "/ZI")
  target_link_options(<target> PUBLIC "/INCREMENTAL")
endif()
```

Attach to a CMake project running on Linux

Visual Studio allows you to debug a process running on a remote Linux system or WSL and debug it with the GDB debugger. To get started, select **Debug** > **Attach to Process...**, set the **Connection type** to **SSH**, and select your **Connection target** from the list of connections in the Connection Manager. Select a process from the list of available processes and press **Attach**. GDB must be installed on your Linux machine. For more information on SSH connections, see the Connection Manager



CMake partial activation

In Visual Studio 2022 version 17.1 and later, CMake functionality won't be enabled automatically if your root folder doesn't contain a `CMakeLists.txt` file. Instead, a dialog will prompt you on whether you'd like to enable CMake functionality for your project. If you decline, CMake cache generation won't start and CMake configurations (from `CMakeSettings.json` or `CMakePresets.json`) won't appear in the configuration dropdown. If you accept, you'll be taken to a workspace-level configuration file, `CMakeWorkspaceSettings.json` (stored in the `.vs` directory), to specify the folders you'd like to enable CMake for. (These folders contain your root `CMakeLists.txt` files).

The accepted properties are:

Expand table

Property	Description
`enableCMake`	Enable Visual Studio's integration for this workspace.

Property	Description
`sourceDirectory`	A string or array of strings specifying the directory or directories with `CMakeLists.txt`. Macros (such as `${workspaceRoot}`) are allowed. Relative paths are based on the workspace root. Directories outside of the current workspace will be ignored.

You can reach `CMakeWorkspaceSettings.json` through the **Project** > **CMake Workspace Settings** menu command at any time, even if CMake functionality is currently disabled.

Open an existing cache

When you open an existing CMake cache file (*`CMakeCache.txt`*), Visual Studio doesn't try to manage your cache and build tree for you. Your custom or preferred tools have complete control over how CMake configures your project.

You can add an existing CMake cache to an open project. It's done the same way you'd add a new configuration. For more information, see our blog post on opening an existing cache in Visual Studio.

> **Note**
>
> The default existing cache experience relies on `cmake-server`, which was removed from CMake in version 3.20. To continue using existing cache functionality in Visual Studio 2019 version 16.10 and later, take one of these steps:
>
> - Manually install CMake version 3.19 or lower. Then, set the `cmakeExecutable` property in your existing cache configuration to use that version of CMake.
> - In your existing cache configuration, set the `cacheGenerationCommand` property to let Visual Studio request the necessary CMake file-based API files. For more information on that property, see CMakeSettings.json reference.
> - Use a query file to request the CMake file-based API files when generating your cache before it's opened in Visual Studio. For query file instructions, see the next section, Advanced CMake cache troubleshooting.

Advanced CMake cache troubleshooting

Visual Studio uses the CMake file-based API (in versions 3.14 and later) to populate the editor with information specific to your project structure. For more information, see the C++ team blog post on multi-root workspaces and file-based API.

Before generating the CMake cache, your custom or preferred tools might need to create a query file named `.cmake/api/v1/query/client-MicrosoftVS/query.json` in your build output folder (the folder that contains `CMakeCache.txt`). The query file should contain this content:

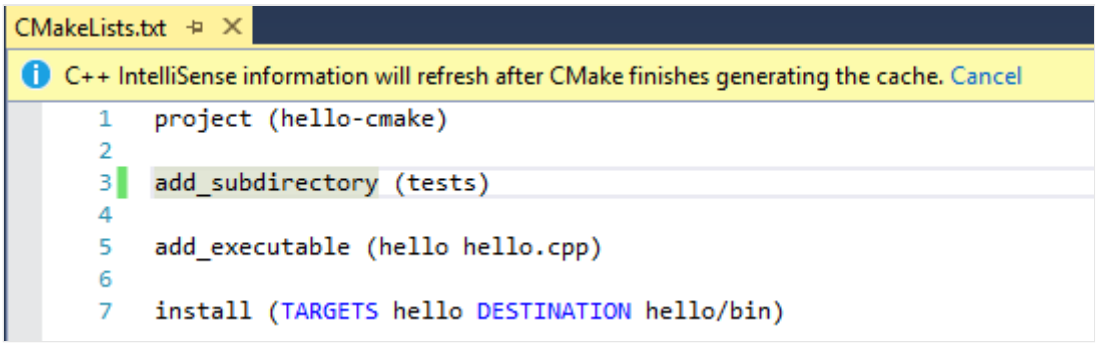
JSON

```
{"requests":[{"kind":"cache","version":2},{"kind":"cmakeFiles","version":1},
{"kind":"codemodel","version":2}]}
```

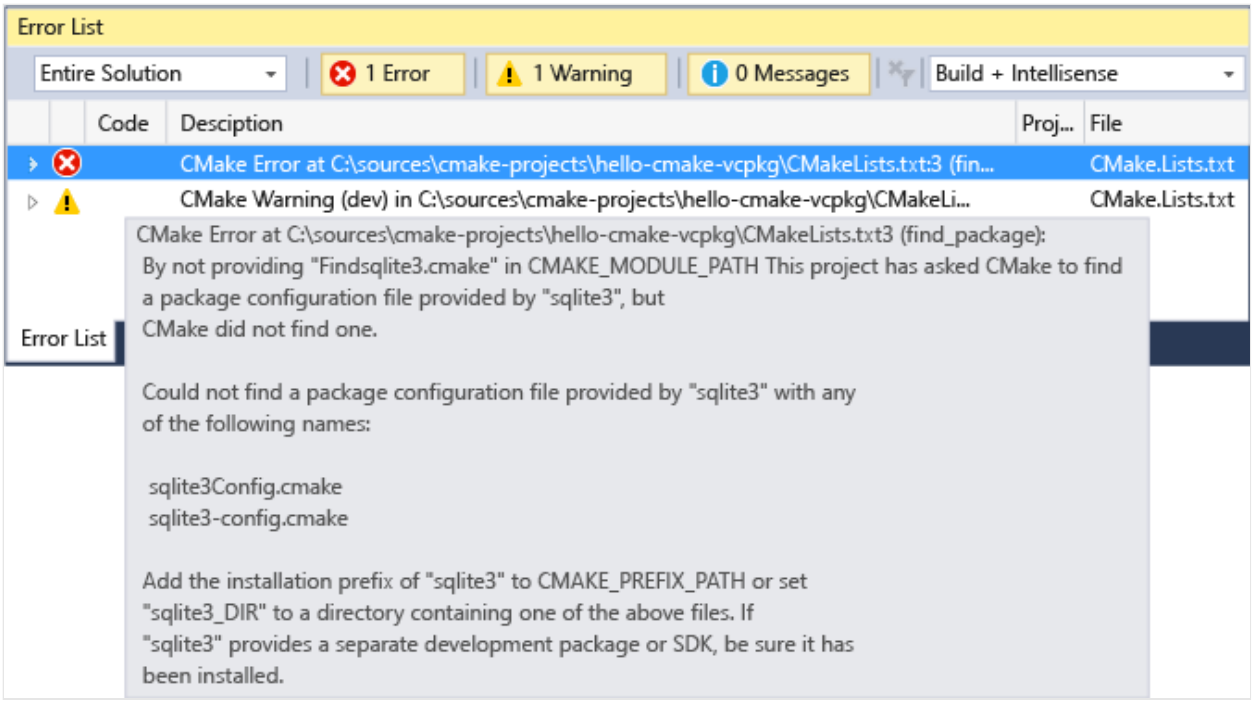
When your custom or preferred tools generate your cache, CMake places files under `.cmake/api/v1/response` that Visual Studio uses to populate the editor with information specific to your project structure.

Editing `CMakeLists.txt` files

To edit a `CMakeLists.txt` file, right-click on the file in **Solution Explorer** and choose **Open**. If you make changes to the file, a yellow status bar appears and informs you that IntelliSense will update. It gives you a chance to cancel the update operation. For information about `CMakeLists.txt`, see the CMake documentation.

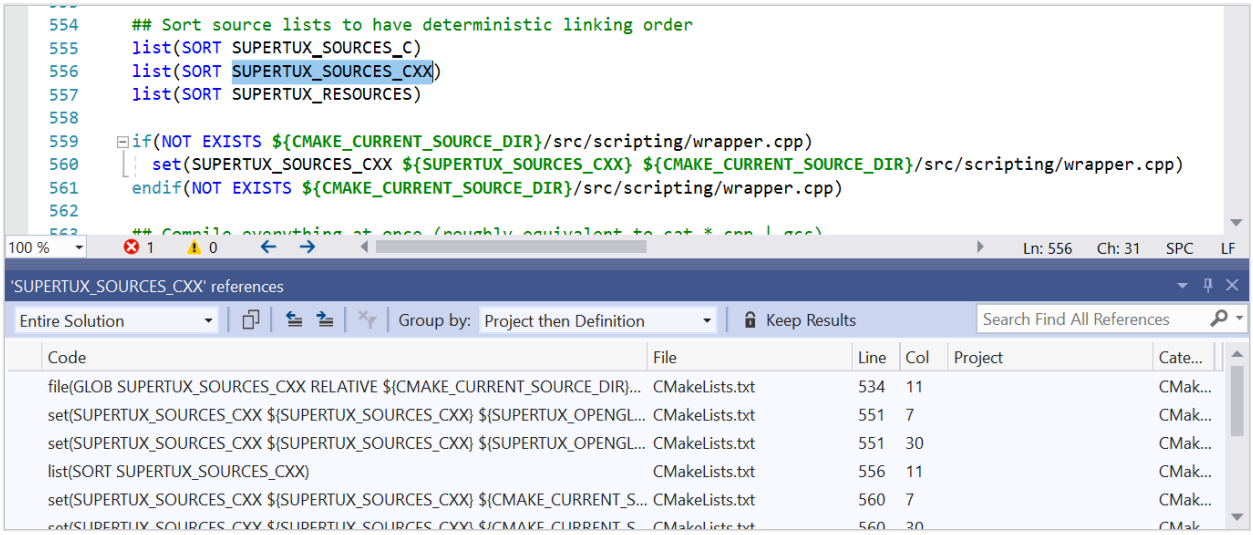


As soon as you save the file, the configuration step automatically runs again and displays information in the **Output** window. Errors and warnings are shown in the **Error List** or **Output** window. Double-click on an error in the **Error List** to navigate to the offending line in `CMakeLists.txt`.



Language services for CMake

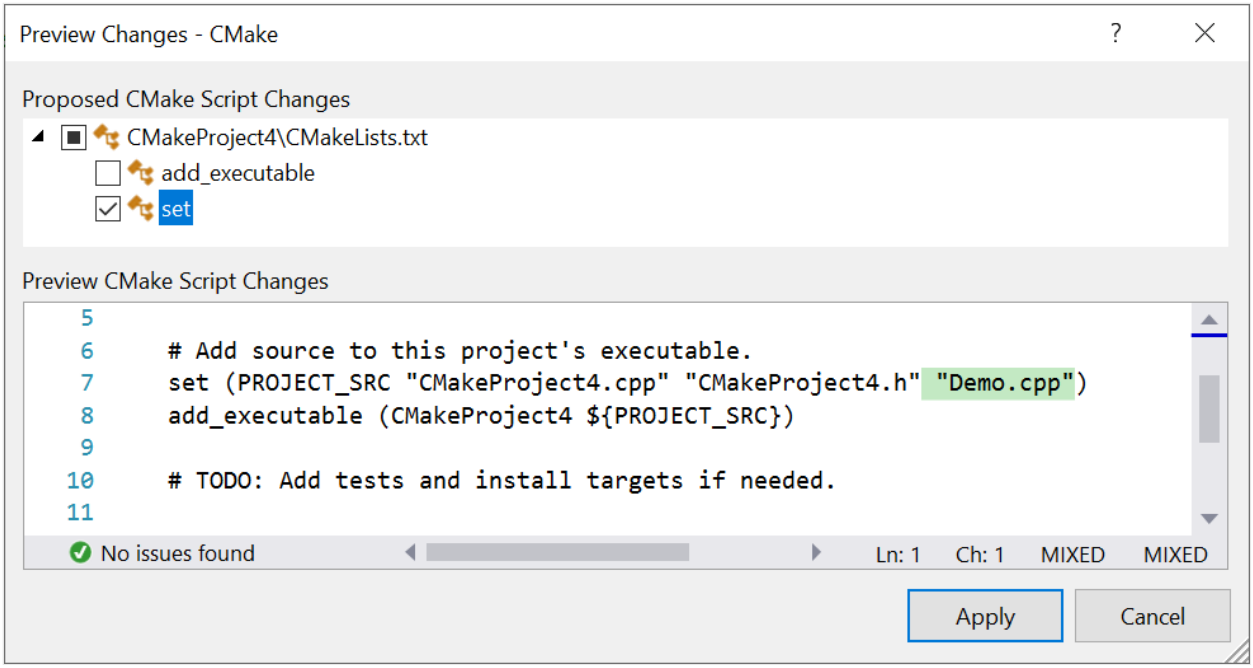
Language services for CMake are available in Visual Studio 2019 version 16.5 or later. It supports code navigation features like Go To Definition, Peek Definition, and Find All References for CMake variables, functions, and targets in CMake script files. For more information, see Code Navigation for CMake Scripts .



CMake project manipulation

CMake project manipulation is available in Visual Studio 2019 version 16.5 or later. Project manipulation enables you to add, remove, and rename source files and targets in your CMake project without manually editing your CMake scripts. When you add or remove files from the Solution Explorer, Visual Studio automatically edits your CMake project. There could be more than one place where it makes sense to add or remove a

reference to a CMake script. If so, Visual Studio asks you where you want to make the change and displays a preview of the proposed changes. For step-by-step instructions, see Add, Remove, and Rename Files and Targets in CMake Projects.



IntelliSense for CMake projects

By default, Visual Studio uses the IntelliSense mode that matches the compiler and target architecture specified by the active CMake configuration.

If `CMakePresets.json` is your active CMake configuration file, then you can specify IntelliSense options using `intelliSenseMode` and `intelliSenseOptions` in the Visual Studio Settings vendor map. For more information, see the Visual Studio Settings vendor map reference.

If `CMakeSettings.json` is your active CMake configuration file, then you can specify IntelliSense options using `intelliSenseMode` in `CMakeSettings.json`. For more information, see the CMakeSettings.json reference.

Configure IntelliSense with CMake toolchain files

In Visual Studio 2019 version 16.9 and later, Visual Studio automatically configures IntelliSense in CMake projects based on CMake variables when you use a CMake toolchain file. For more information, see Configure IntelliSense with CMake Toolchain Files.

Vcpkg integration

CMake projects opened in Visual Studio integrate with vcpkg, a cross-platform C/C++ dependency manager. Before using vcpkg with Visual Studio, you must run `vcpkg integrate install`. For instructions and more information about vcpkg, see:

- Install and use packages with CMake in Visual Studio
- vcpkg in CMake projects

If `CMakeSettings.json` is your active configuration file, Visual Studio automatically passes the vcpkg toolchain file (`vcpkg.cmake`) to CMake. This behavior is disabled automatically when you specify any other toolchain in your CMake Settings configuration.

If `CMakePresets.json` is your active configuration file, you'll need to set the path to `vcpkg.cmake` in `CMakePresets.json`. We recommend using the `VCPKG_ROOT` environment variable instead of an absolute path to keep the file shareable. For more information, see Enable vcpkg integration with CMake Presets. `CMakePresets.json` is available in Visual Studio 2019 version 16.10 or later and is the recommended CMake configuration file.

Run CMake from the command line

If `CMakePresets.json` is your active CMake configuration file, then you can easily reproduce your local builds outside of Visual Studio. For more information, see Run CMake from the command line or a CI pipeline. `CMakePresets.json` is supported in Visual Studio 2019 version 16.10 or later and is the recommended CMake configuration file.

If `CMakeSettings.json` is your active CMake configuration file, then you'll need to manually pass the arguments that are encoded in your `CMakeSettings.json` file to CMake. If you have installed CMake from the Visual Studio Installer, you can run it from the command line by following these steps:

1. Run the appropriate `vsdevcmd.bat` file (x86/x64). For more information, see Building on the command line .
2. Switch to your output folder.
3. Run CMake to build or configure your app.

See also

Tutorial: Create C++ cross-platform projects in Visual Studio
Configure a Linux CMake project
Connect to your remote Linux computer
Customize CMake build settings
CMakeSettings.json schema reference
Configure CMake debugging sessions
Deploy, run, and debug your Linux project
CMake predefined configuration reference
vcpkg in CMake projects
Install and use packages with CMake in Visual Studio

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Tutorial: Create C++ cross-platform projects in Visual Studio

Article • 10/16/2023

Visual Studio C and C++ development isn't just for Windows anymore. This tutorial shows how to use Visual Studio for C++ cross platform development on Windows and Linux. It's based on CMake, so you don't have to create or generate Visual Studio projects. When you open a folder that contains a CMakeLists.txt file, Visual Studio configures the IntelliSense and build settings automatically. You can quickly start editing, building, and debugging your code locally on Windows. Then, switch your configuration to do the same on Linux, all from within Visual Studio.

In this tutorial, you learn how to:

- ✓ clone an open-source CMake project from GitHub
- ✓ open the project in Visual Studio
- ✓ build and debug an executable target on Windows
- ✓ add a connection to a Linux machine
- ✓ build and debug the same target on Linux

Prerequisites

- Set up Visual Studio for Cross Platform C++ Development
 - First, install Visual Studio and choose the **Desktop development with C++** and **Linux development with C++ workloads**. This minimal install is only 3 GB. Depending on your download speed, installation shouldn't take more than 10 minutes.

- Set up a Linux machine for Cross Platform C++ Development

 - Visual Studio doesn't require any specific distribution of Linux. The OS can be running on a physical machine, in a VM, or in the cloud. You could also use the Windows Subsystem for Linux (WSL). However, for this tutorial, a graphical environment is required. WSL isn't recommended here, because it's intended primarily for command-line operations.

 - Visual Studio requires these tools on the Linux machine: C++ compilers, `gdb`, `ssh`, `rsync`, `make`, and `zip`. On Debian-based systems, you can use this command to install these dependencies:

Windows Command Prompt

```
sudo apt install -y openssh-server build-essential gdb rsync make
zip
```

- Visual Studio requires a recent version of CMake on the Linux machine that has server mode enabled (at least 3.8). Microsoft produces a universal build of CMake that you can install on any Linux distro. We recommend you use this build to ensure that you have the latest features. You can get the CMake binaries from the Microsoft fork of the CMake repo on GitHub. Go to that page and download the version that matches the system architecture on your Linux machine, then mark it as an executable:

 Windows Command Prompt

  ```
  wget <path to binary>
  chmod +x cmake-3.11.18033000-MSVC_2-Linux-x86_64.sh
  ```

- You can see the options for running the script with `--help`. We recommend that you use the `-prefix` option to specify installing in the **/usr** path, because **/usr/bin** is the default location where Visual Studio looks for CMake. The following example shows the Linux-x86_64 script. Change it as needed if you're using a different target platform.

 Windows Command Prompt

  ```
  sudo ./cmake-3.11.18033000-MSVC_2-Linux-x86_64.sh --skip-license --
  prefix=/usr
  ```

- Git for windows installed on your Windows machine.
- A GitHub account.

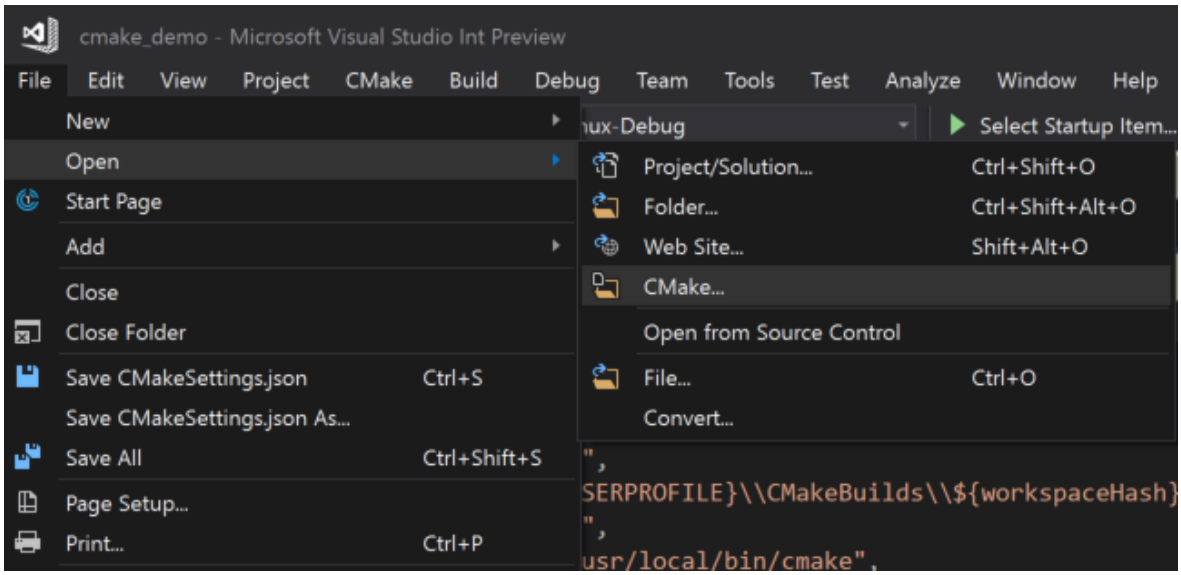
Clone an open-source CMake project from GitHub

This tutorial uses the Bullet Physics SDK on GitHub. It provides collision detection and physics simulations for many applications. The SDK includes sample executable programs that compile and run without having to write other code. This tutorial doesn't modify any of the source code or build scripts. To start, clone the *bullet3* repository from GitHub on the machine where you have Visual Studio installed.

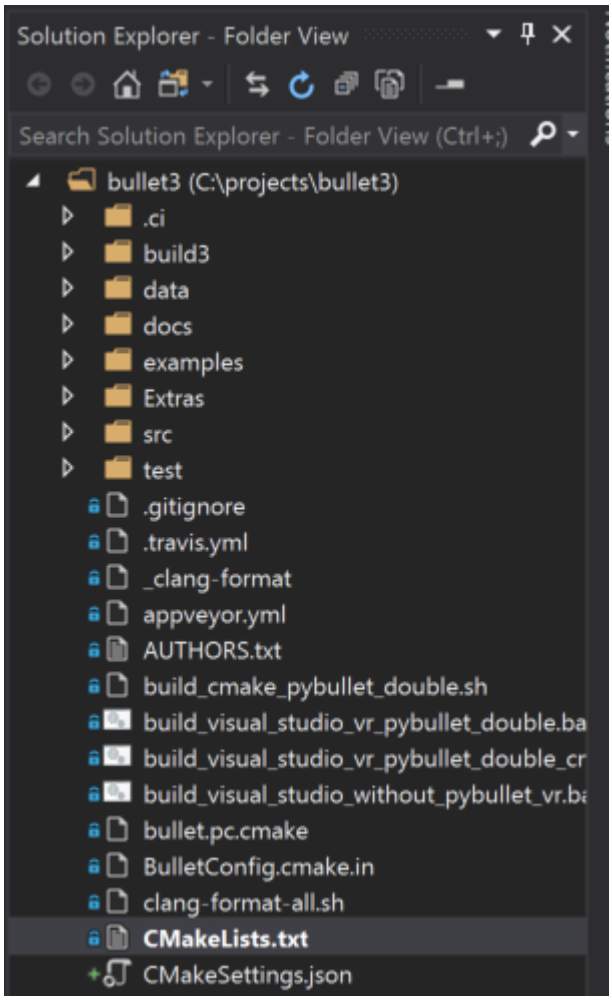
Windows Command Prompt

```
git clone https://github.com/bulletphysics/bullet3.git
```

1. On the Visual Studio main menu, choose **File** > **Open** > **CMake**. Navigate to the `CMakeLists.txt` file in the root of the bullet3 repo you downloaded.

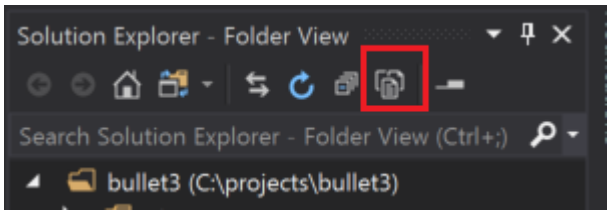


As soon as you open the folder, your folder structure becomes visible in the **Solution Explorer**.



This view shows you exactly what is on disk, not a logical or filtered view. By default, it doesn't show hidden files.

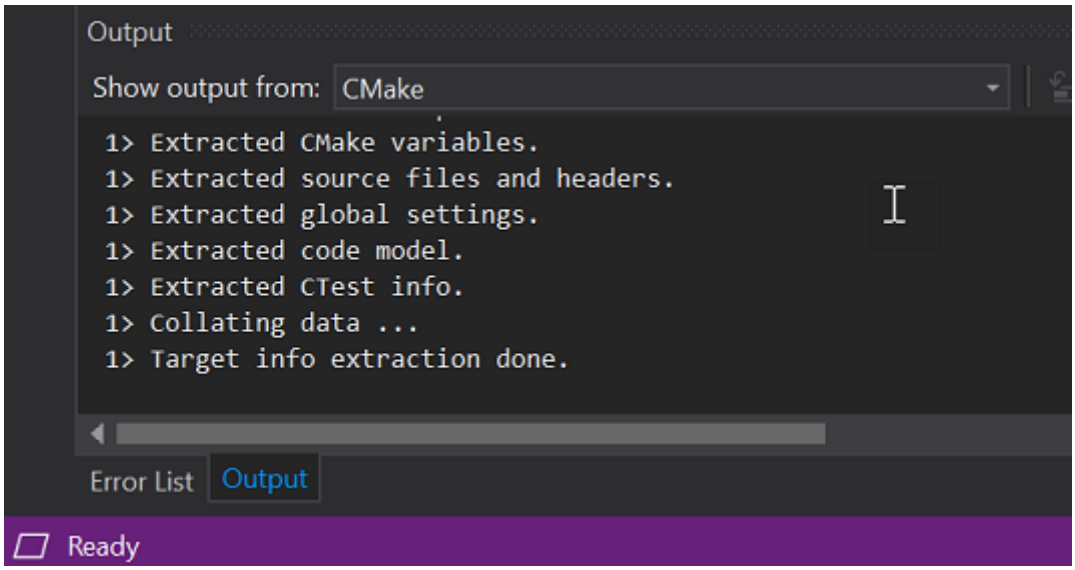
2. Choose the **Show all files** button to see all the files in the folder.



Switch to targets view

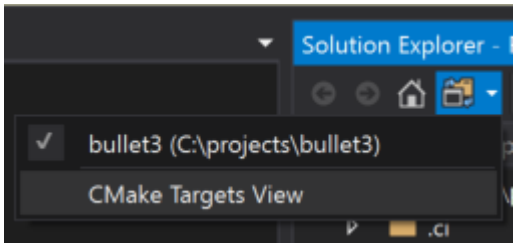
When you open a folder that uses CMake, Visual Studio automatically generates the CMake cache. This operation might take a few moments, depending on the size of your project.

1. In the **Output Window**, select **Show output from** and then choose **CMake** to monitor the status of the cache generation process. When the operation is complete, it says "Target info extraction done."

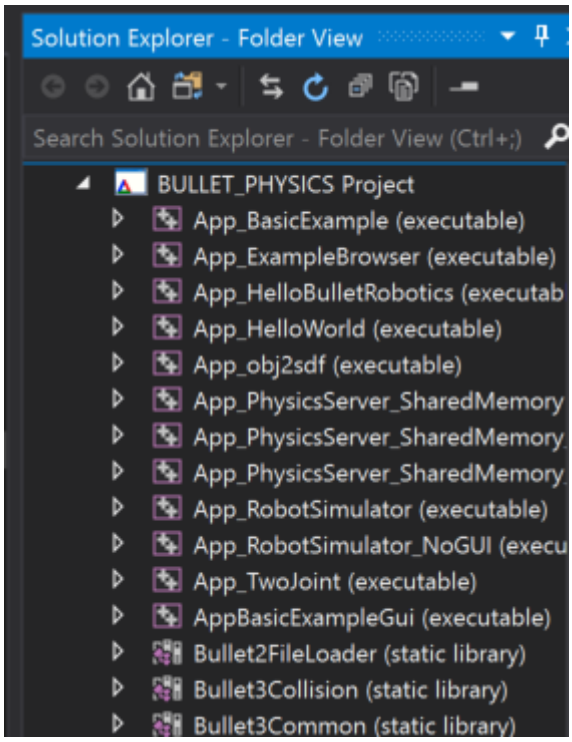


After this operation completes, IntelliSense is configured. You can build the project, and debug the application. Visual Studio now shows a logical view of the solution, based on the targets specified in the CMakeLists files.

2. Use the **Solutions and Folders** button in the **Solution Explorer** to switch to CMake Targets View.



Here's what that view looks like for the Bullet SDK:



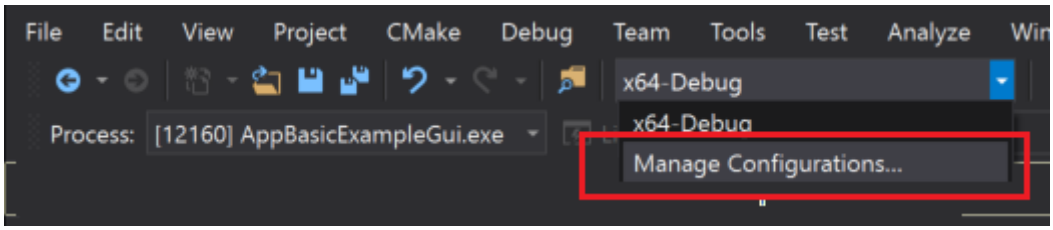
Targets view provides a more intuitive view of what is in this source base. You can see some targets are libraries and others are executables.

3. Expand a node in CMake Targets View to see its source code files, wherever those files might be located on disk.

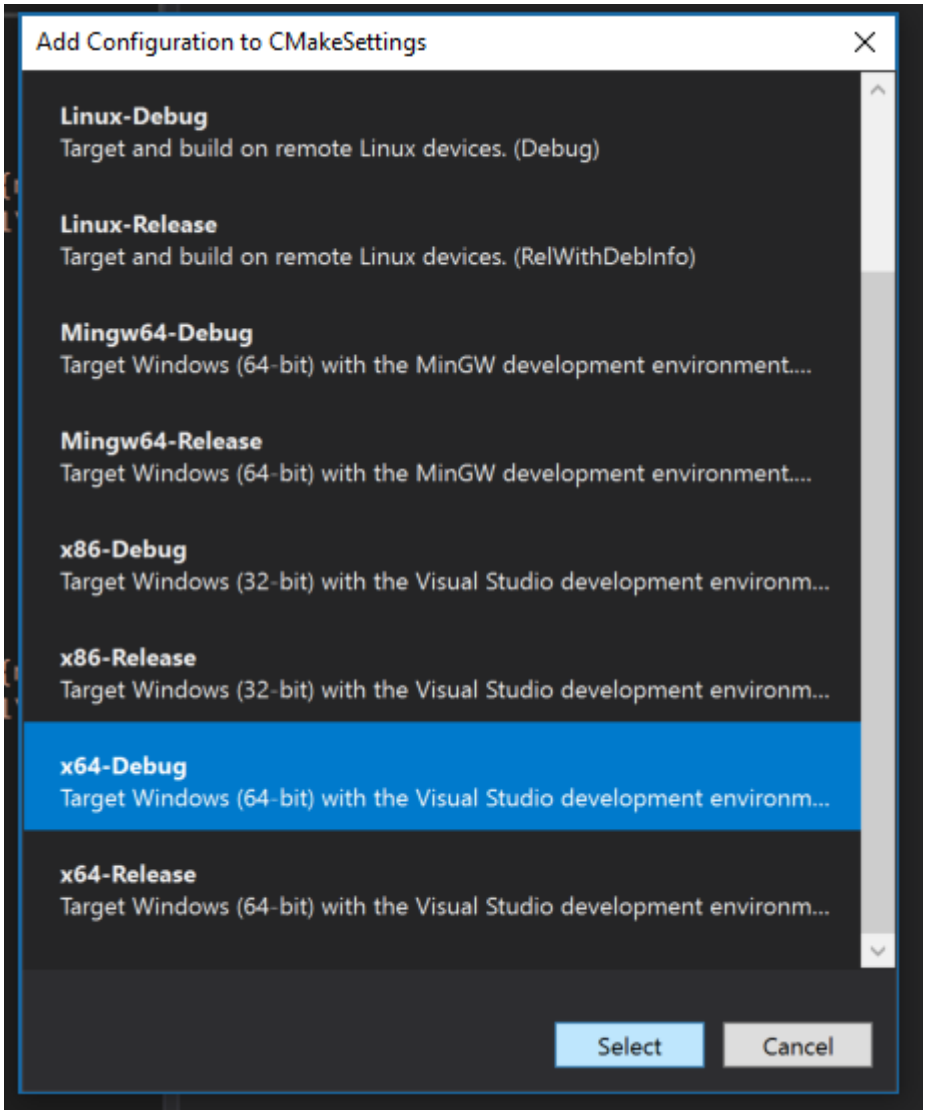
Add an explicit Windows x64-Debug configuration

Visual Studio creates a default **x64-Debug** configuration for Windows. Configurations are how Visual Studio understands what platform target it's going to use for CMake. The default configuration isn't represented on disk. When you explicitly add a configuration, Visual Studio creates a file called *CMakeSettings.json*. It's populated with settings for all the configurations you specify.

1. Add a new configuration. Open the **Configuration** drop-down in the toolbar and select **Manage Configurations**.



The CMake Settings Editor opens. Select the green plus sign on the left-hand side of the editor to add a new configuration. The **Add Configuration to CMakeSettings** dialog appears:



This dialog shows all the configurations included with Visual Studio, plus any custom configurations that you create. If you want to continue to use a **x64-Debug** configuration that should be the first one you add. Select **x64-Debug**, and then choose the **Select** button. Visual Studio creates the CMakeSettings.json file with a configuration for **x64-Debug**, and saves it to disk. You can use whatever names you like for your configurations by changing the name parameter directly in CMakeSettings.json.

Set a breakpoint, build, and run on Windows

In this step, we debug an example program that demonstrates the Bullet Physics library.

1. In **Solution Explorer**, select AppBasicExampleGui and expand it.

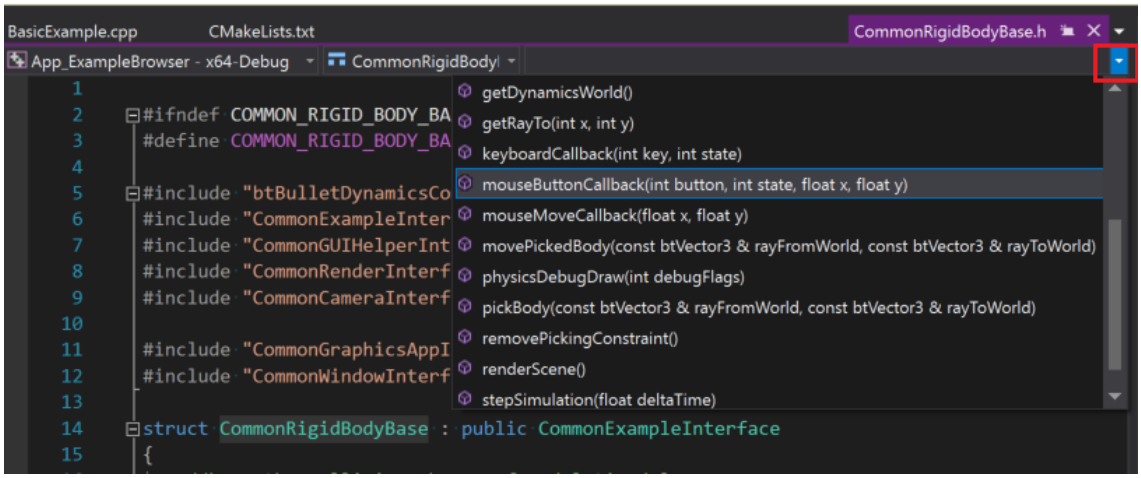
2. Open the file `BasicExample.cpp`.

3. Set a breakpoint that gets hit when you click in the running application. The click event is handled in a method within a helper class. To quickly get there:

 a. Select `CommonRigidBodyBase` that the struct `BasicExample` is derived from. It's around line 30.

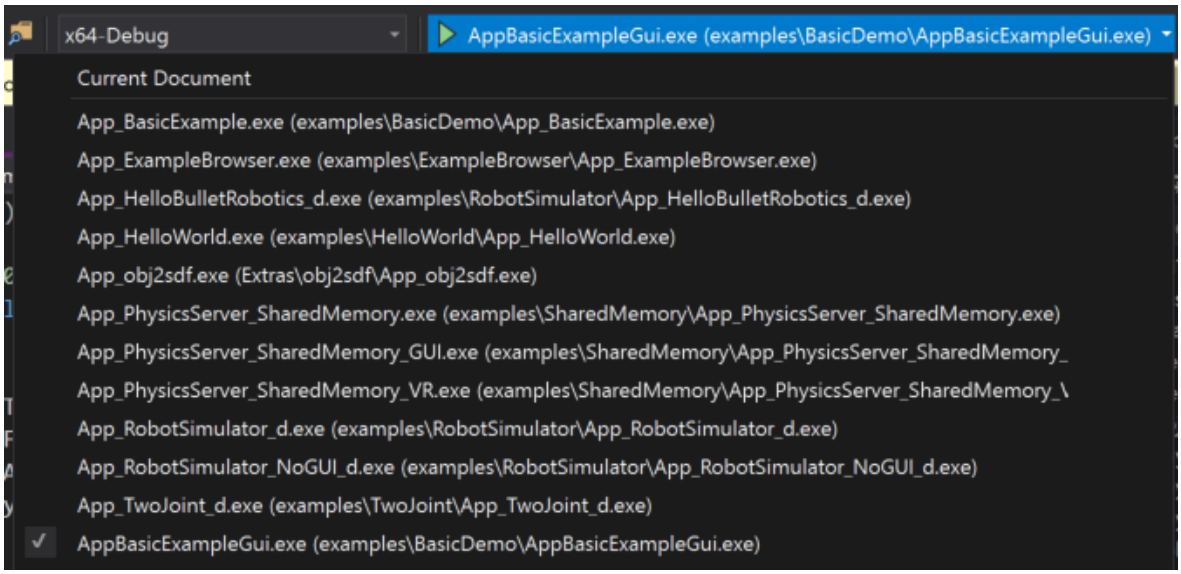
 b. Right-click and choose **Go to Definition**. Now you're in the header CommonRigidBodyBase.h.

 c. In the browser view above your source, you should see that you're in the `CommonRigidBodyBase`. To the right, you can select members to examine. Open the drop-down and select `mouseButtonCallback` to go to the definition of that function in the header.

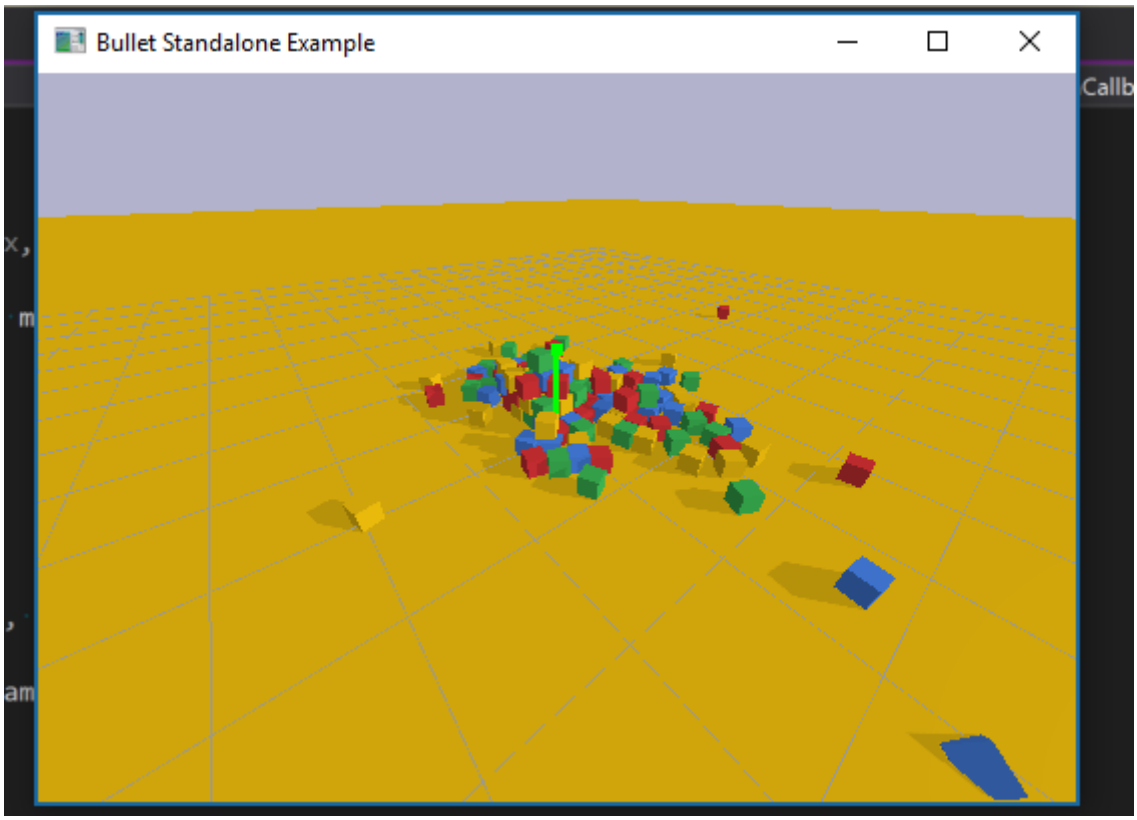


4. Place a breakpoint on the first line within this function. It gets hit when you click a mouse button within the window of the application, when run under the Visual Studio debugger.

5. To launch the application, select the launch drop-down in the toolbar. It's the one with the green play icon that says "Select Startup Item." In the drop-down, select AppBasicExampleGui.exe. The executable name now displays on the launch button:



6. Choose the launch button to build the application and necessary dependencies, then launch it with the Visual Studio debugger attached. After a few moments, the running application appears:



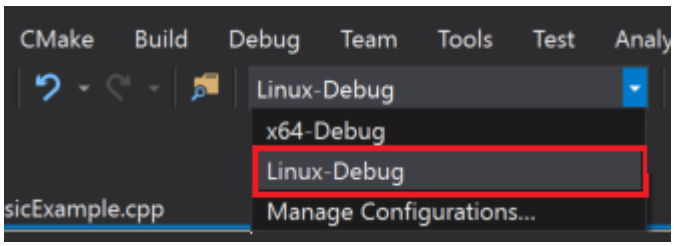
7. Move your mouse into the application window, then click a button to trigger the breakpoint. The breakpoint brings Visual Studio back to the foreground, and the editor shows the line where execution is paused. You can inspect the application variables, objects, threads, and memory, or step through your code interactively. Choose **Continue** to let the application resume, and then exit it normally. Or, halt execution within Visual Studio by using the stop button.

Add a Linux configuration and connect to the remote machine

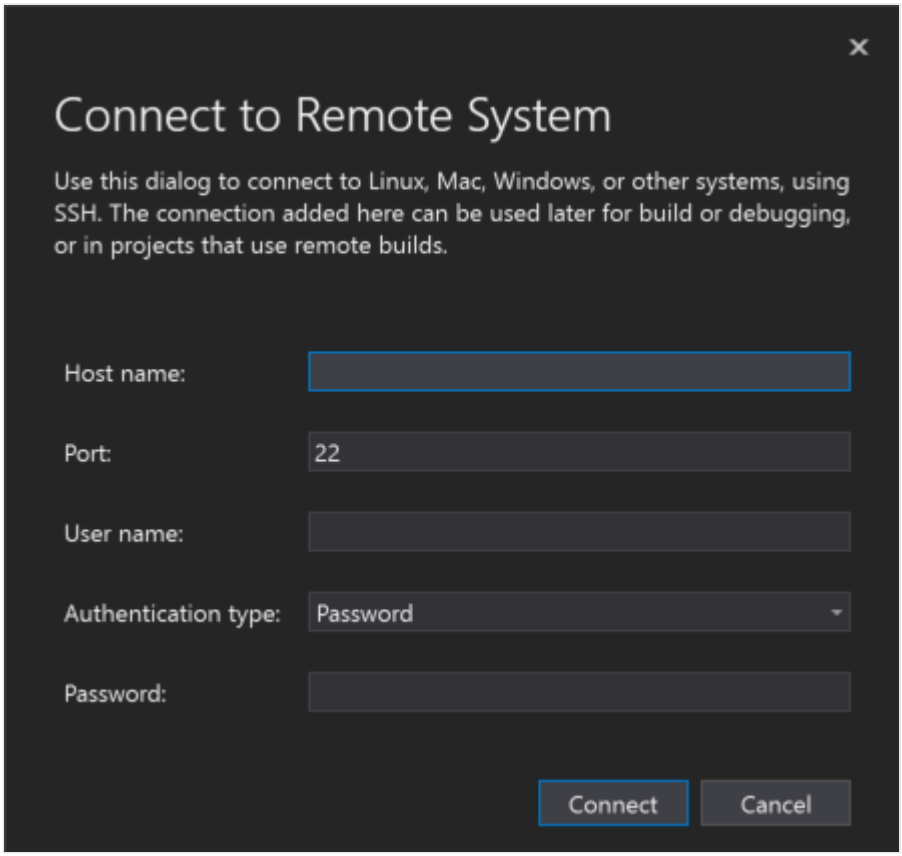
1. Add a Linux configuration. Right-click the CMakeSettings.json file in the **Solution Explorer** view and select **Add Configuration**. You see the same Add Configuration to CMakeSettings dialog as before. Select **Linux-Debug** this time, then save the CMakeSettings.json file (ctrl + s).

2. **Visual Studio 2019 version 16.6 or later** Scroll down to the bottom of the CMake Settings Editor and select **Show advanced settings**. Select **Unix Makefiles** as the **CMake generator**, then save the CMakeSettings.json file (ctrl + s).

3. Select **Linux-Debug** in the configuration drop-down.



If it's the first time you're connecting to a Linux system, the **Connect to Remote System** dialog appears.



If you've already added a remote connection, you can open this window by navigating to **Tools** > **Options** > **Cross Platform** > **Connection Manager**.

4. Provide the connection information to your Linux machine and choose **Connect**. Visual Studio adds that machine as to CMakeSettings.json as your default connection for **Linux-Debug**. It also pulls down the headers from your remote machine, so you get IntelliSense specific to that remote connection. Next, Visual Studio sends your files to the remote machine and generates the CMake cache on the remote system. These steps might take some time, depending on the speed of your network and power of your remote machine. You know it's complete when the message "Target info extraction done" appears in the CMake output window.

Set a breakpoint, build, and run on Linux

Because it's a desktop application, you need to provide some more configuration information to the debug configuration.

1. In the CMake Targets view, right-click AppBasicExampleGui and choose **Debug and Launch Settings** to open the launch.vs.json file that's in the hidden **.vs** subfolder. This file is local to your development environment. You can move it into the root of your project if you wish to check it in and save it with your team. In this file, a configuration has been added for AppBasicExampleGui. These default settings work in most cases, but not here. Because it's a desktop application, you need to provide some additional information to launch the program so you can see it on your Linux machine.

2. To find the value of the environment variable `DISPLAY` on your Linux machine, run this command:

 Windows Command Prompt

   ```
   echo $DISPLAY
   ```

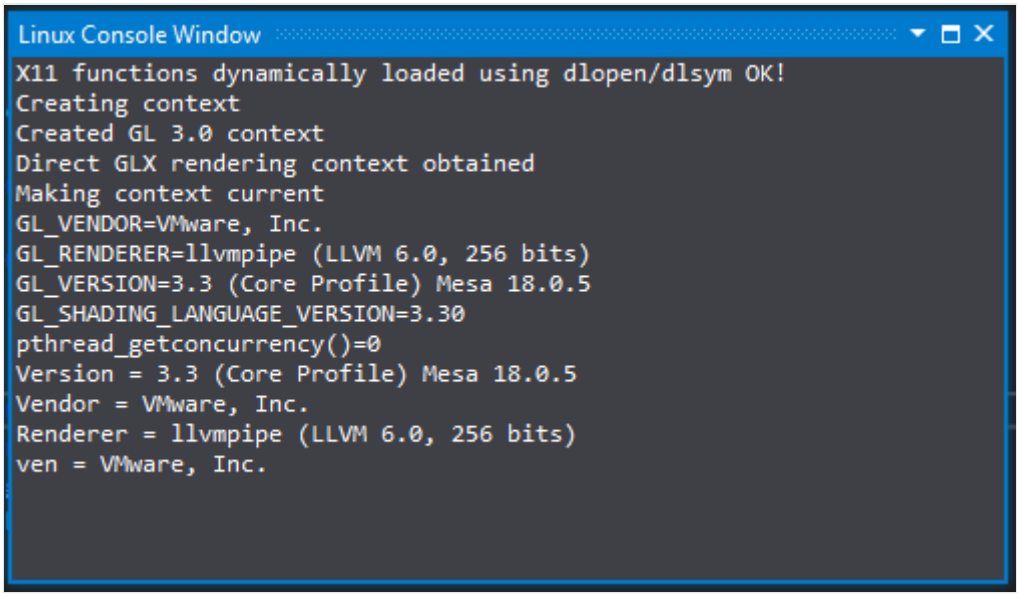
 In the configuration for AppBasicExampleGui, there's a parameter array, "pipeArgs". It contains a line: "${debuggerCommand}". It's the command that launches `gdb` on the remote machine. Visual Studio must export the display into this context before that command runs. For example, if the value of your display is `:1`, modify that line as follows:

 Windows Command Prompt

```
"export DISPLAY=:1;${debuggerCommand}",
```

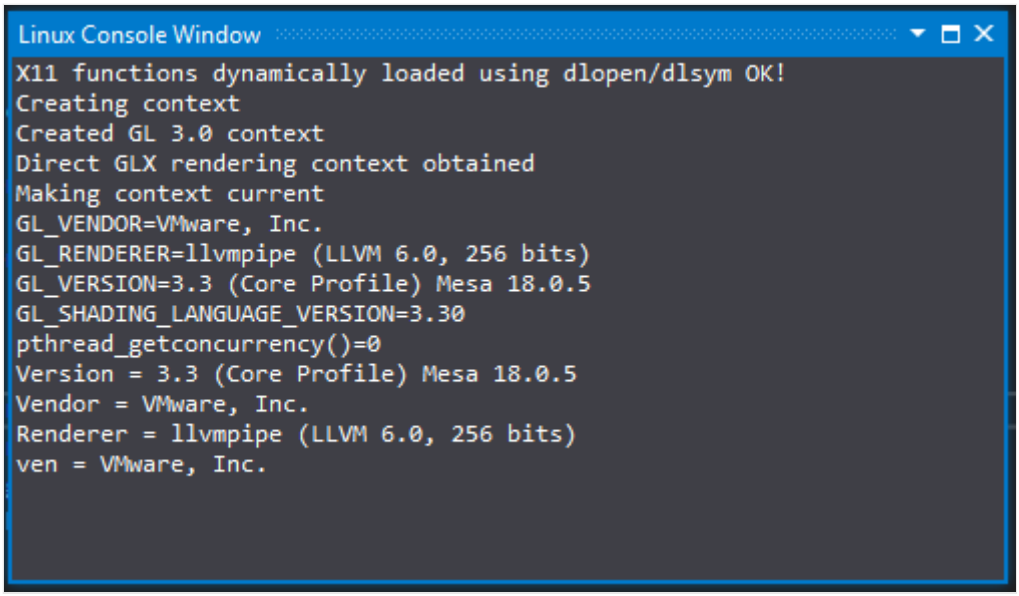
3. Launch and debug your application. Open the **Select Startup Item** drop-down in the toolbar and choose **AppBasicExampleGui**. Next, either choose the green play icon in the toolbar, or press **F5**. The application and its dependencies are built on the remote Linux machine, then launched with the Visual Studio debugger attached. On your remote Linux machine, you should see an application window appear.

4. Move your mouse into the application window, and click a button. The breakpoint is hit. Program execution pauses, Visual Studio comes back to the foreground, and you see your breakpoint. You should also see a Linux Console Window appear in Visual Studio. The window provides output from the remote Linux machine, and it can also accept input for `stdin`. Like any Visual Studio window, you can dock it where you prefer to see it. Its position is persisted in future sessions.



5. You can inspect the application variables, objects, threads, memory, and step through your code interactively using Visual Studio. But this time, you're doing it all on a remote Linux machine instead of your local Windows environment. You can choose **Continue** to let the application resume and exit normally, or you can choose the stop button, as with local execution.

6. Look at the Call Stack window and view the Calls to `x11OpenGLWindow` since Visual Studio launched the application on Linux.



What you learned

In this tutorial, you cloned a code base directly from GitHub. You built, ran, and debugged it on Windows without modifications. Then you used the same code base, with minor configuration changes, to build, run, and debug on a remote Linux machine.

Next steps

Learn more about configuring and debugging CMake projects in Visual Studio:

- CMake Projects in Visual Studio
- Configure a Linux CMake project
- Connect to your remote Linux computer
- Customize CMake build settings
- Configure CMake debugging sessions
- Deploy, run, and debug your Linux project
- CMake predefined configuration reference

Walkthrough: Build and debug C++ with WSL 2 and Visual Studio 2022

Article • 03/20/2024

Visual Studio 2022 introduces a native C++ toolset for Windows Subsystem for Linux version 2 (WSL 2) development. This toolset is available now in Visual Studio 2022 version 17.0 or higher.

WSL 2 is the new, recommended version of the Windows Subsystem for Linux (WSL). It provides better Linux file system performance, GUI support, and full system call compatibility. Visual Studio's WSL 2 toolset allows you to use Visual Studio to build and debug C++ code on WSL 2 distros without adding an SSH connection. You can already build and debug C++ code on WSL 1 distros using the native WSL 1 toolset introduced in Visual Studio 2019 version 16.1.

Visual Studio's WSL 2 toolset supports both CMake and MSBuild-based Linux projects. CMake is our recommendation for all C++ cross-platform development with Visual Studio. We recommend CMake because it build and debug the same project on Windows, WSL, and remote systems.

For a video presentation of the information in this topic, see Video: Debug C++ with WSL 2 Distributions and Visual Studio 2022.

WSL 2 toolset background

C++ cross-platform support in Visual Studio assumes all source files originate in the Windows file system. When targeting a WSL 2 distro, Visual Studio executes a local `rsync` command to copy files from the Windows file system to the WSL file system. The local `rsync` copy doesn't require any user intervention. It occurs automatically when Visual Studio detects you're using a WSL 2 distro. To learn more about the differences between WSL 1 and WSL 2, see Comparing WSL 1 and WSL 2.

CMake Presets integration in Visual Studio supports the WSL 2 toolset. To learn more, see CMake Presets integration in Visual Studio and Visual Studio Code and Configure and build with CMake Presets in Visual Studio. There's also more advanced information in this article under Advanced WSL 2 and CMake projects considerations.

Install the build tools

Install the tools necessary to build and debug on WSL 2. You'll install a recent version of CMake using Visual Studio's CMake binary deployment in a later step.

1. Install WSL and a WSL 2 distro by following the instructions at Install WSL.

2. Assuming your distro uses `apt` (this walkthrough uses Ubuntu), use the following commands to install the required build tools on your WSL 2 distro:

   ```
   Bash

   sudo apt update
   sudo apt install g++ gdb make ninja-build rsync zip
   ```

 The `apt` commands above install:

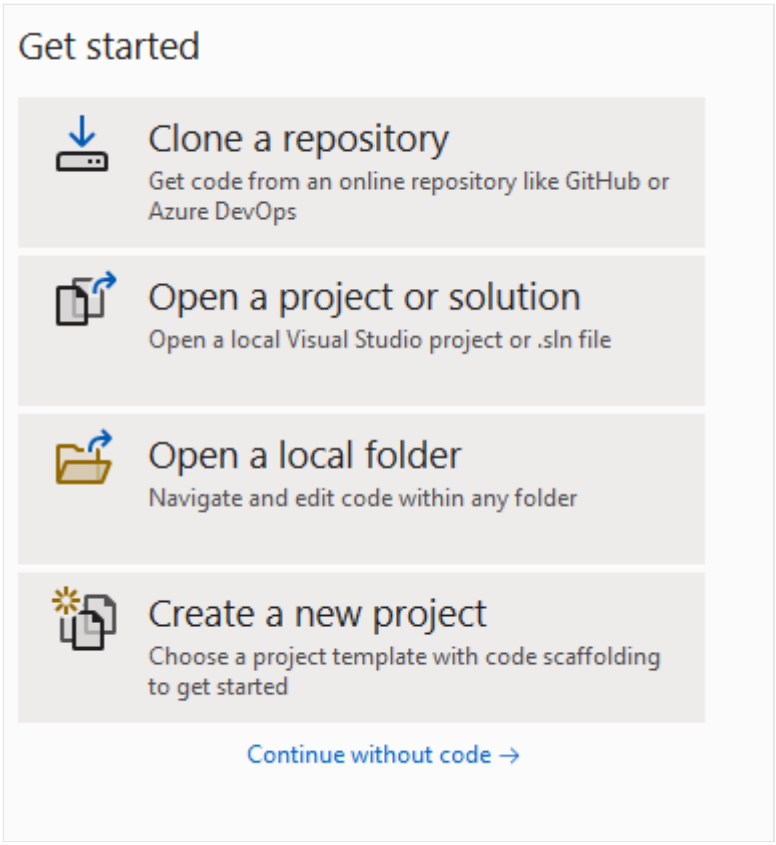
 - A C++ compiler
 - `gdb`
 - `CMake`
 - `rsync`
 - `zip`
 - An underlying build system generator

Cross-platform CMake development with a WSL 2 distro

This walkthrough uses GCC and Ninja on Ubuntu. And Visual Studio 2022 version 17.0 Preview 2 or later.

Visual Studio defines a CMake project as a folder with a `CMakeLists.txt` file at the project root. In this walkthrough, you create a new CMake project by using the Visual Studio **CMake Project** template:

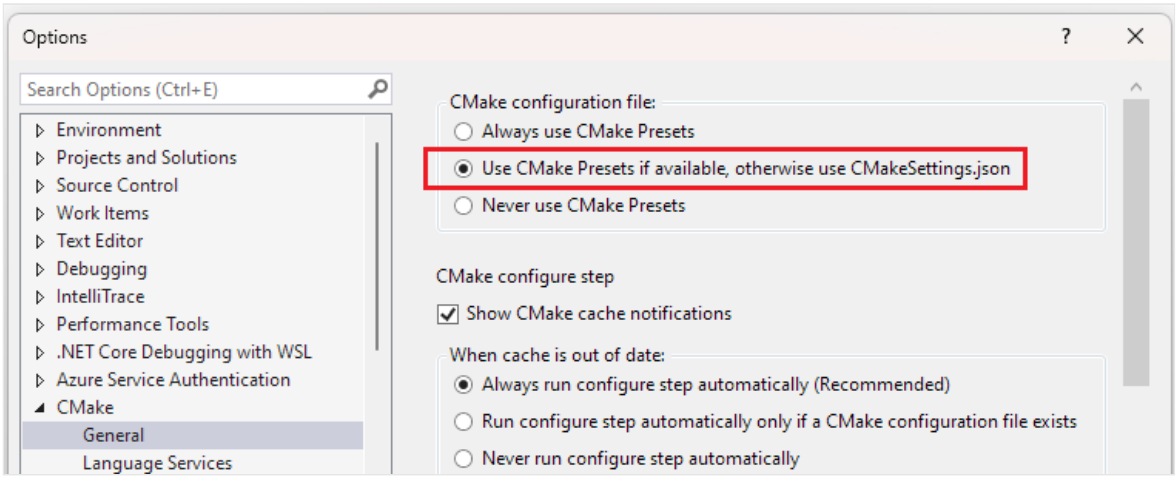
3. From the Visual Studio **Get started** screen, select **Create a new project**.



The available options are: Clone a repository, Open a project or solution, Open a local folder, Create a new project, or Continue without code."::::

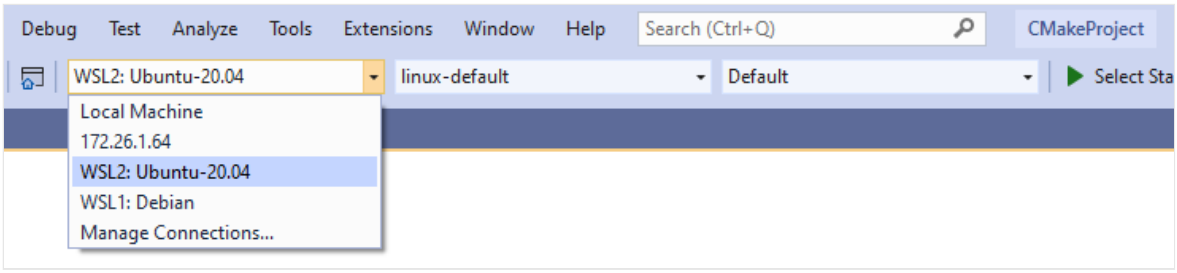
4. In the **Search for templates** textbox, type "cmake". Choose the **CMake Project** type and select **Next**. Give the project a name and location, and then select **Create**.

5. Enable Visual Studio's CMake Presets integration. Select **Tools** > **Options** > **CMake** > **General**. Select **Prefer using CMake Presets for configure, build, and test**, then select **OK**. Instead, you could have added a `CMakePresets.json` file to the root of the project. For more information, see Enable CMake Presets integration.



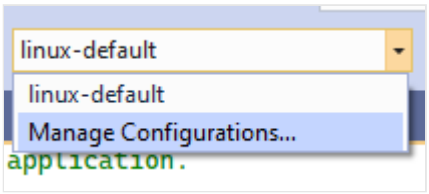
6. To activate the integration: from the main menu, select **File** > **Close Folder**. The **Get started** page appears. Under **Open recent**, select the folder you just closed to reopen the folder.

7. There are three dropdowns across the Visual Studio main menu bar. Use the dropdown on the left to select your active target system. This is the system where CMake is invoked to configure and build the project. Visual Studio queries for WSL installations with `wsl -l -v`. In the following image, **WSL2: Ubuntu-20.04** is shown selected as the **Target System**.



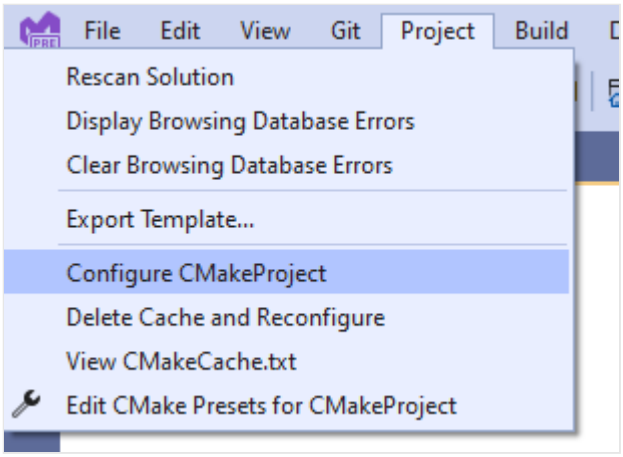
> **Note**
>
> If Visual Studio starts to configure your project automatically, read step 11 to manage CMake binary deployment, and then continue to the step below. To customize this behavior, see Modify automatic configuration and cache notifications.

8. Use the dropdown in the middle to select your active Configure Preset. Configure Presets tell Visual Studio how to invoke CMake and generate the underlying build system. In step 7, the active Configure Preset is the **linux-default** Preset created by Visual Studio. To create a custom Configure Preset, select **Manage Configurations...** For more information about Configure Presets, see Select a Configure Preset and Edit Presets.

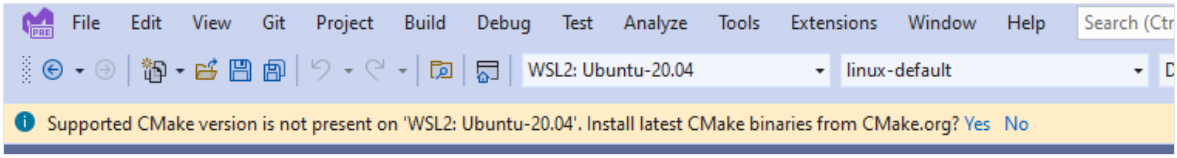


9. Use the dropdown on the right to select your active Build Preset. Build Presets tell Visual Studio how to invoke build. In the illustration for step 7, the active Build Preset is the **Default** Build Preset created by Visual Studio. For more information about Build Presets, see Select a Build Preset.

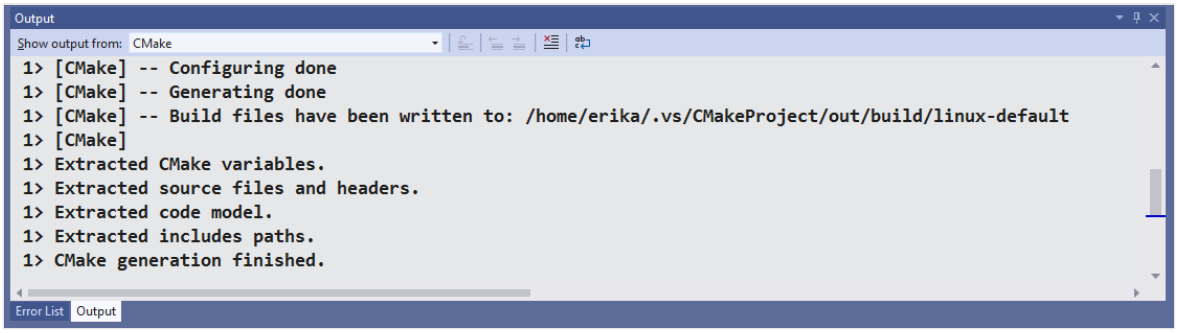
10. Configure the project on WSL 2. If project generation doesn't start automatically, then manually invoke configure with **Project** > **Configure** *project-name*



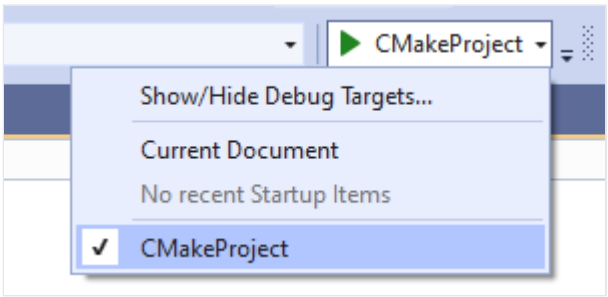
11. If you don't have a supported version of CMake installed on your WSL 2 distro, then Visual Studio prompts you beneath the main menu ribbon to deploy a recent version of CMake. Select **Yes** to deploy CMake binaries to your WSL 2 distro.



12. Confirm that the configure step completed and that you can see the **CMake generation finished** message in the **Output** window under the **CMake** pane. Build files are written to a directory in the WSL 2 distro's file system.

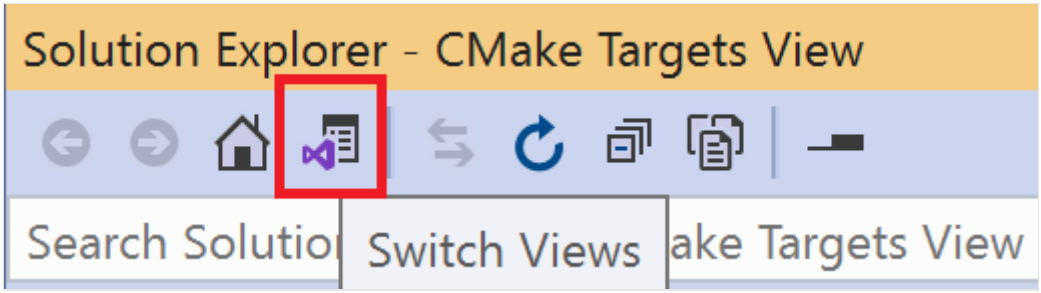


13. Select the active debug target. The debug dropdown menu lists all the CMake targets available to the project.

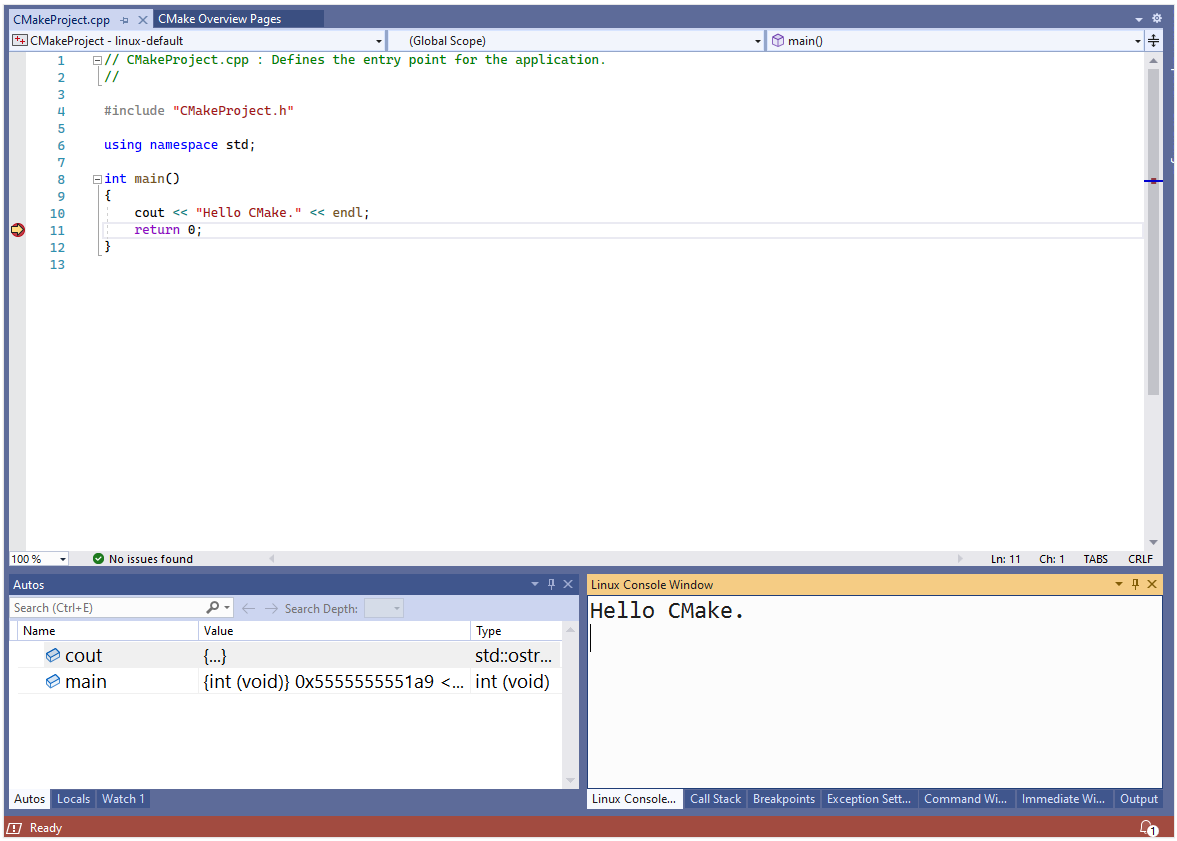


14. Expand the project subfolder in the **Solution Explorer**. In the `CMakeProject.cpp` file, set a breakpoint in `main()`. You can also navigate to CMake targets view by

selecting the View Picker button in the **Solution Explorer**, highlighted in following screenshot:



15. Select **Debug** > **Start**, or press **F5**. Your project builds, the executable launches on your WSL 2 distro, and Visual Studio halts execution at the breakpoint. The output of your program (in this case, `"Hello CMake."`) is visible in the Linux Console Window:



You've now built and debugged a C++ app with WSL 2 and Visual Studio 2022.

Advanced WSL 2 and CMake projects considerations

Visual Studio only provides native support for WSL 2 for CMake projects that use `CMakePresets.json` as the active configuration file. To migrate from `CMakeSettings.json` to `CMakePresets.json`, see Enable CMake Presets integration in Visual Studio.

If you're targeting a WSL 2 distribution and you don't want to use the WSL 2 toolset, then in the Visual Studio Remote Settings vendor map in `CMakePresets.json`, set **forceWSL1Toolset** to **true** . For more information, see Visual Studio Remote Settings vendor map.

If **forceWSL1Tooslet** is set to **true**, then Visual Studio doesn't maintain a copy of your source files in the WSL file system. Instead, it accesses source files in the mounted Windows drive (`/mnt/`...).

In most cases, it's best to use the WSL 2 toolset with WSL 2 distributions because WSL 2 is slower when project files are instead stored in the Windows file system. To learn more about file system performance in WSL 2, see Comparing WSL 1 and WSL 2.

Specify advanced settings such as the path to the directory on WSL 2 where the project is copied, copy source options, and rsync command arguments, in the Visual Studio Remote Settings vendor map in `CMakePresets.json`. For more information, see Visual Studio Remote Settings vendor map.

System headers are still automatically copied to the Windows file system to supply the native IntelliSense experience. You can customize the headers that are included or excluded from this copy in the Visual Studio Remote Settings vendor map in `CMakePresets.json`.

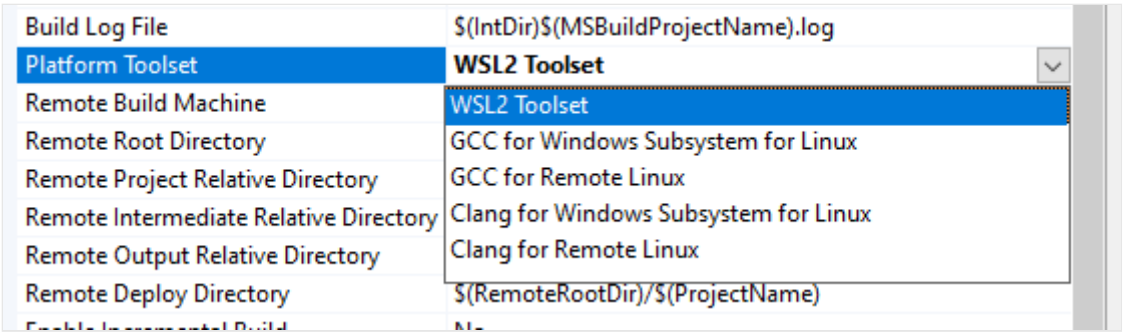
You can change the IntelliSense mode, or specify other IntelliSense options, in the Visual Studio Settings vendor map in `CMakePresets.json`. For details about the vendor map, see Visual Studio Remote Settings vendor map.

WSL 2 and MSBuild-based Linux projects

CMake is recommended for all C++ cross-platform development with Visual Studio because it allows you to build and debug the same project on Windows, WSL, and remote systems.

But you may have a MSBuild-based Linux project.

If you have a MSBuild-based Linux project, then you can upgrade to the WSL 2 toolset in Visual Studio. Right-click the project in the solution explorer, then choose **Properties** > **General** > **Platform Toolset**:



If you're targeting a WSL 2 distribution and you don't want to use the WSL 2 toolset, then in the **Platform Toolset** dropdown, select the **GCC for Windows Subsystem for Linux** or **Clang for Windows Subsystem for Linux** toolset. If either of these toolsets are selected, Visual Studio doesn't maintain a copy of your source files in the WSL file system and instead accesses source files over the mounted Windows drive (`/mnt/`...). System headers are still automatically copied to the Windows file system to provide a native IntelliSense experience. Customize the headers that are included or excluded from this copy in **Property Pages** > **General**.

In most cases, it's best to use the WSL 2 toolset with WSL 2 distributions because WSL 2 is slower when project files are stored in the Windows file system. To learn more, see Comparing WSL 1 and WSL 2.

See also

Video: Debug C++ with WSL 2 Distributions and Visual Studio 2022
Download Visual Studio 2022
Create a CMake Linux project in Visual Studio
Tutorial: Debug a CMake project on a remote Windows machine

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Tutorial: Debug a CMake project on a remote Windows machine

Article • 03/11/2022

This tutorial uses Visual Studio C++ on Windows to create and build a CMake project that you can deploy and debug on a remote Windows machine. The tutorial is specific to Windows ARM64, but the steps can be generalized for other architectures.

In Visual Studio, the default debugging experience for ARM64 is remote debugging an ARM64 Windows machine. Configure your debug settings as shown in this tutorial. Otherwise, when you try to debug an ARM64 CMake project, you'll get an error that Visual Studio can't find the remote machine.

In this tutorial, you'll learn how to:

- ✓ create a CMake project
- ✓ configure a CMake project to build for ARM64
- ✓ configure a CMake project to run on a remote ARM64 Windows machine
- ✓ debug a CMake project running on a remote ARM64 Windows machine

Prerequisites

On the host machine

To set up Visual Studio for cross-platform C++ development, install the build tools for the target architecture. For this tutorial, install the ARM64 build tools by doing these steps:

1. Run the Visual Studio Installer. If you haven't installed Visual Studio yet, see Install Visual Studio
2. On the Visual Studio Installer home screen, choose **Modify**.
3. From the choices at the top, choose **Individual components**.
4. Scroll down to the **Compilers, build tools, and runtimes** section.
5. Ensure that the following are selected:

 - **C++ CMake tools for Windows**
 - **MSVC v142 - VS 2019 C++ ARM64 build tools (Latest)** It's important that you choose the `ARM64` build tools and not the `ARM` build tools (look for the 64) and that you choose the version that goes with `VS 2019`.

6. Select **Modify** to install the tools.

On the remote machine

1. Install the remote tools on the remote machine. For this tutorial, install the ARM64 tools by following the instructions in Download and Install the remote tools.
2. Start and configure the remote debugger on the remote machine. For this tutorial, do so by following the directions in set up the remote debugger on the remote Windows machine.

Create a CMake project

On the Windows host machine:

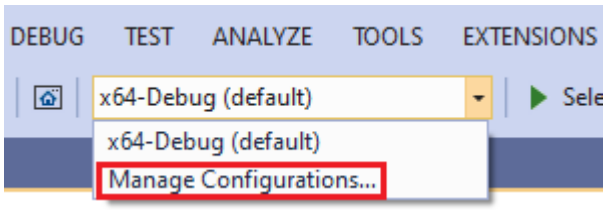
1. Run Visual Studio
2. From the main menu, select **File** > **New** > **Project**.
3. Select **CMake Project** > **Next**
4. Give the project a name and choose a location. Then select **Create**.

Give Visual Studio a few moments to create the project and populate the **Solution Explorer**.

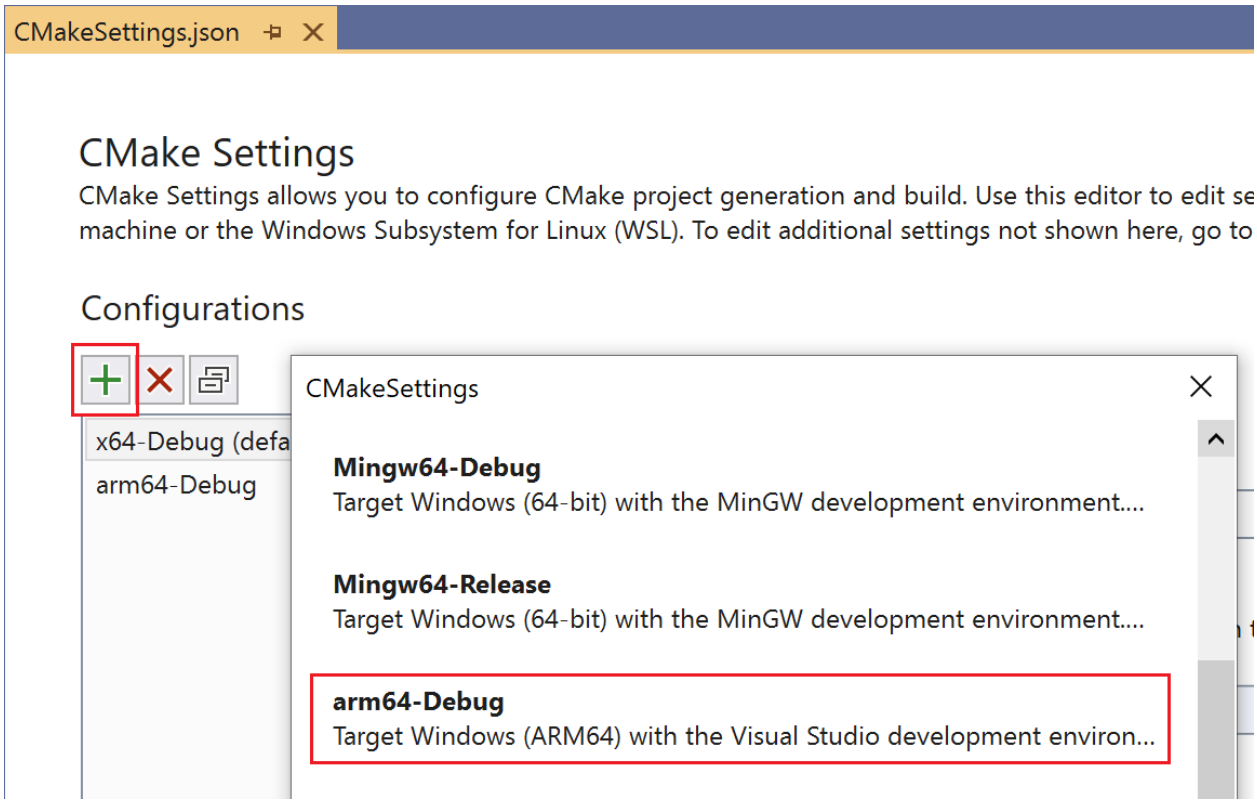
Configure for ARM64

To target an ARM64 Windows machine, you need to build using ARM64 build tools.

Select the Visual Studio **Configuration** dropdown and select **Manage Configurations**.

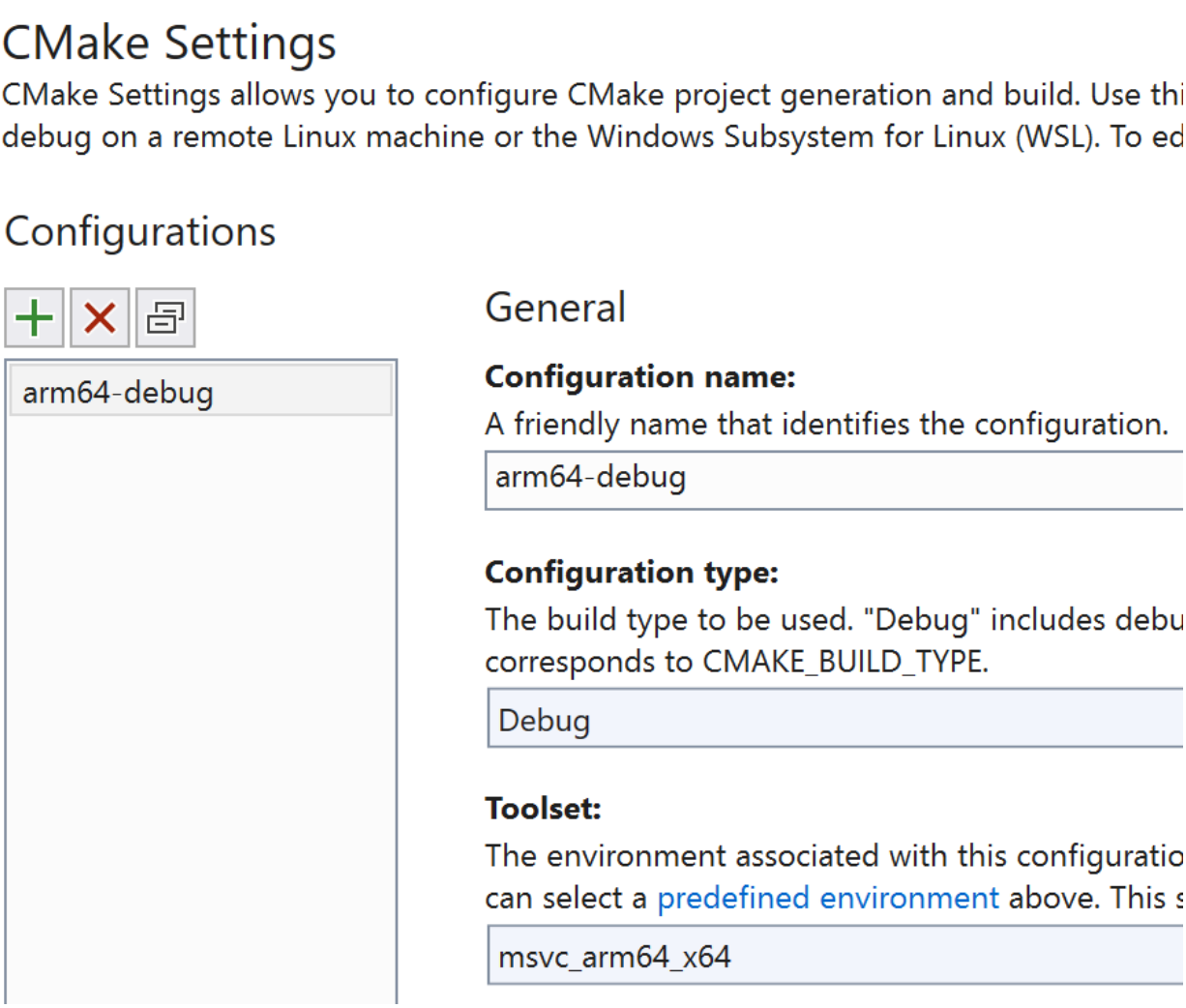


Add a new configuration by selecting **Add a new configuration** (the green + button).
In the **CMakeSettings** dialog that appears, select **arm64-debug**, and then choose **Select**:



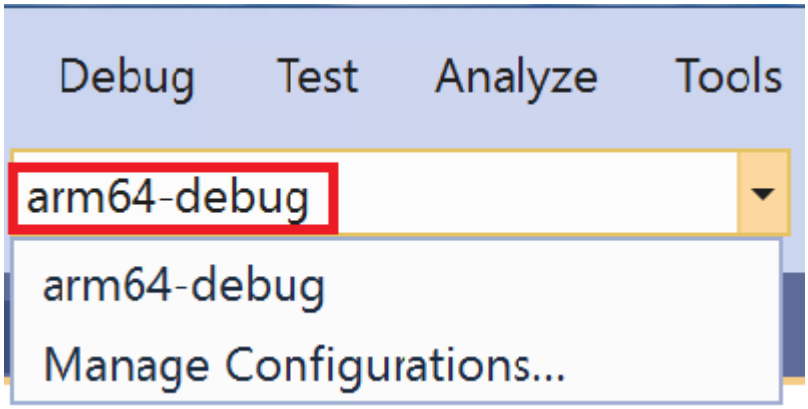
This command adds a debug configuration named `arm64-Debug` to your `CmakeSettings.json` file. This configuration name is a unique, friendly name that makes it easier for you to identify these settings in the **Configuration** dropdown.

The **Toolset** dropdown is set to **msvc_arm64_x64**. Your settings should now look like this:



> ⓘ Note
>
> In the **Toolset** dropdown, **msvc_arm64** selects 32-bit host tools to cross-compile to ARM64, whereas **msvc_arm64 x64** selects 64-bit host tools to cross-compile to ARM64, which is what you'll do in this tutorial. For more information about the available toolset environments, see **Pre-defined environments**.

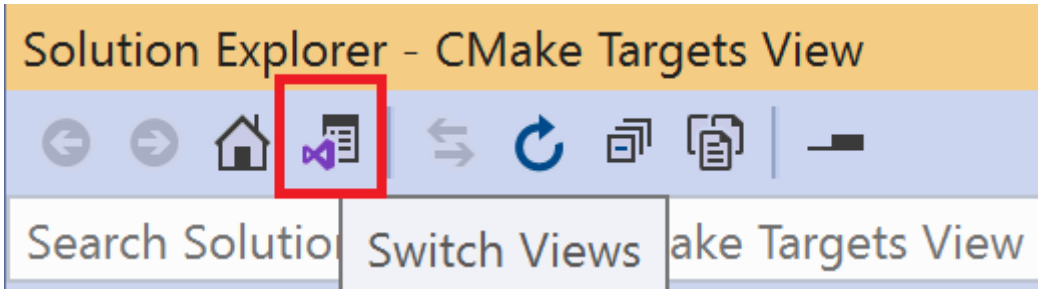
Save the `CMakeSettings.json` file. In the configuration dropdown, select **arm64-debug**. (It may take a moment after saving the `CMakeSettings.json` file for it to appear in the list):



Add a debug configuration file

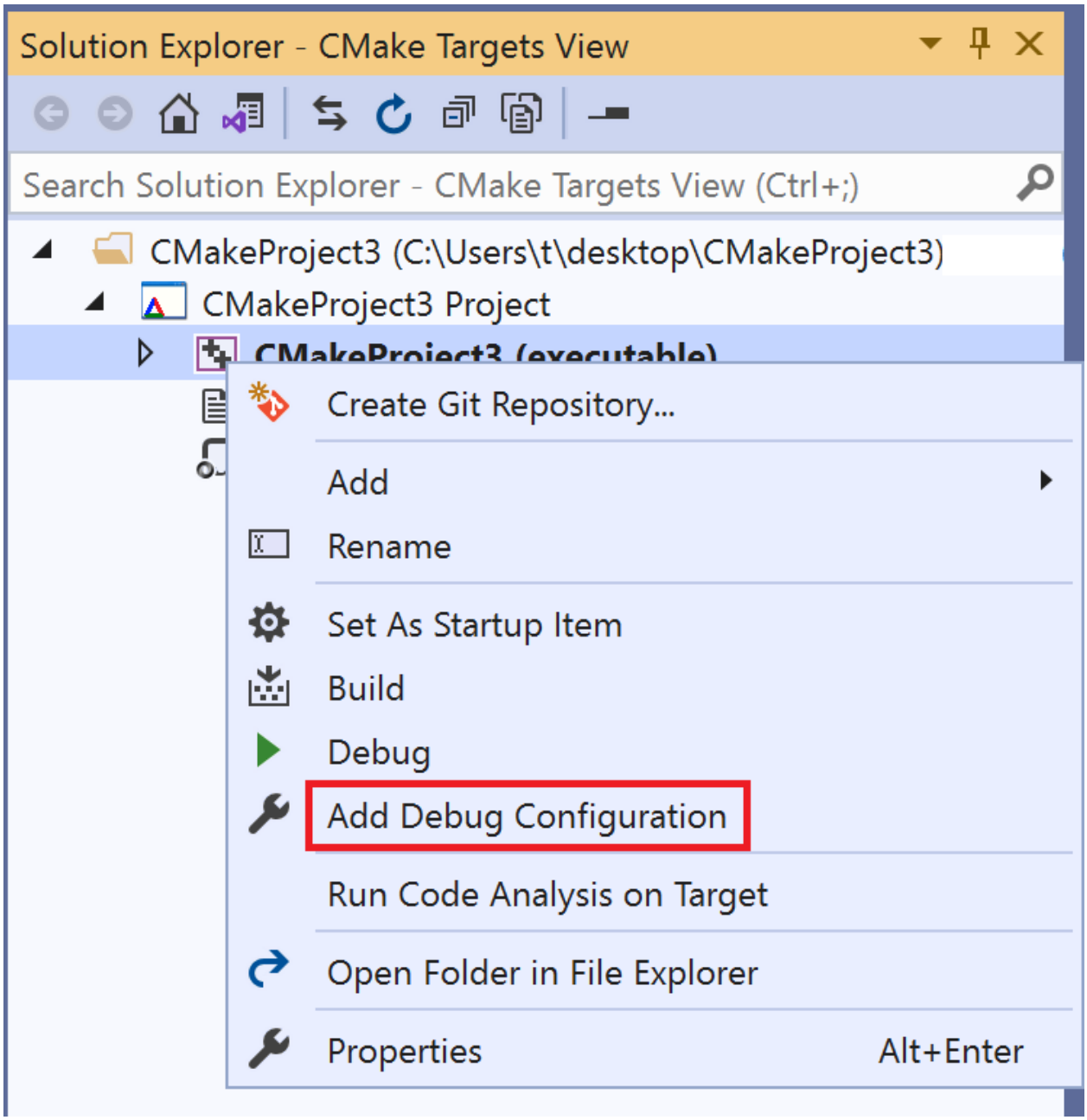
Next, add configuration information that tells Visual Studio where to find your remote machine, along with other configuration details.

Change the **Solution Explorer** view to targets view by selecting the **Switch Views** button:



Then, in the **Solution Explorer**, double-click **CMake Targets View** to see the project.

Open the project folder (in this example, **CMakeProject3 Project**), and then right-click the executable and select **Add Debug Configuration**:



This command creates a `launch.vs.json` file in your project. Open it and change the following entries to enable remote debugging:

- `projectTarget`: this value is set for you if you added the debug configuration file from the **Solution Explorer** targets view per the instructions above.
- `remoteMachineName`: set to the IP address of the remote ARM64 machine, or its machine name.

For more information about `launch.vs.json` settings, see launch.vs.json schema reference.

> **Note**
>
> If you're using the folder view instead of the targets view in **Solution Explorer**, right-click the `CMakeLists.txt` file and select **Add Debug Configuration**. This

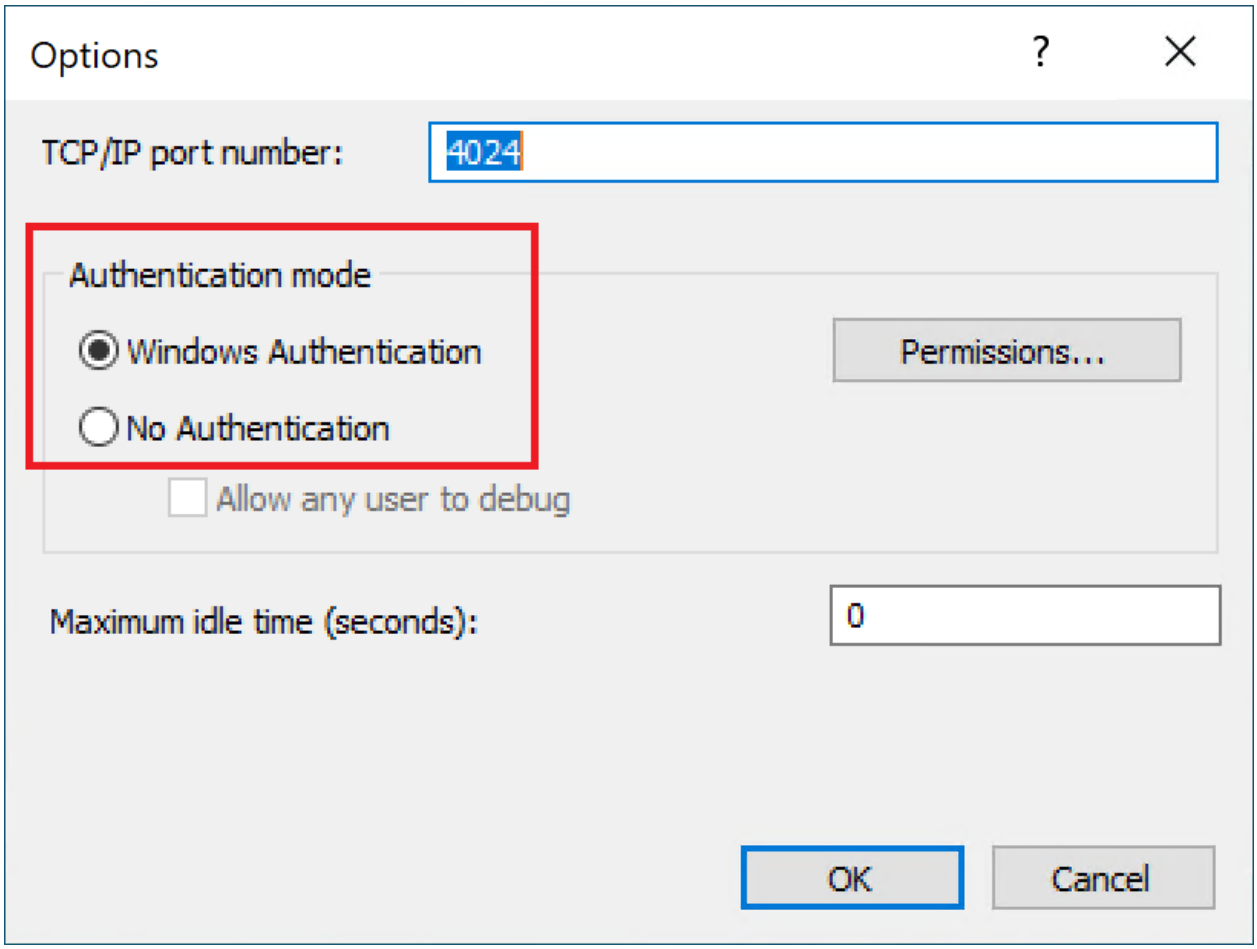
experience differs from adding the debug configuration from the targets view in the following ways:

- You'll be asked to select a debugger (select **C/C++ Remote Windows Debug**).
- Visual Studio will provide less configuration template information in the `launch.vs.json` file so you'll need to add it yourself. You'll need to provide the `remoteMachineName` and `projectTarget` entries. When you add the configuration from the targets view, you only need to specify `remoteMachineName`.
- For the `projectTarget` setting value, check the startup item dropdown to get the unique name of your target, for example, in this tutorial it is `CMakeProject3.exe`'.

Start the remote debugger monitor on the remote Windows machine

Before you run your CMake project, ensure that the Visual Studio 2019 remote debugger is running on the remote Windows machine. You may need to change the remote debugger options depending on your authentication situation.

For example, on the remote machine, from the Visual Studio Remote Debugger menu bar, select **Tools** > **Options**. Set the **authentication mode** to match how your environment is set up:



Then, in Visual Studio on the host machine, update the `launch.vs.json` file to match. For example, if you choose **No Authentication** on the remote debugger, update the `launch.vs.json` file in your project by adding `"authenticationType": "none"` to the `configurations` section `launch.vs.json`. Otherwise, `"authenticationType"` defaults to `"windows"` and doesn't need to be explicitly stated. This example shows a `launch.vs.json` file configured for no authentication:

XAML

```
{
  "version": "0.2.1",
  "defaults": {},
  "configurations": [
    {
        "type": "remoteWindows",
        "authenticationType": "none"
        "name": "CMakeLists.txt",
        "project": "CMakeLists.txt",
        "projectTarget": "CMakeProject3.exe",
        "remoteMachineName": "<ip address goes here>",
        "cwd": "${debugInfo.defaultWorkingDirectory}",
        "program": "${debugInfo.fullTargetPath}",
        "deploy": [],
        "args": [],
        "env": {}
    },
```

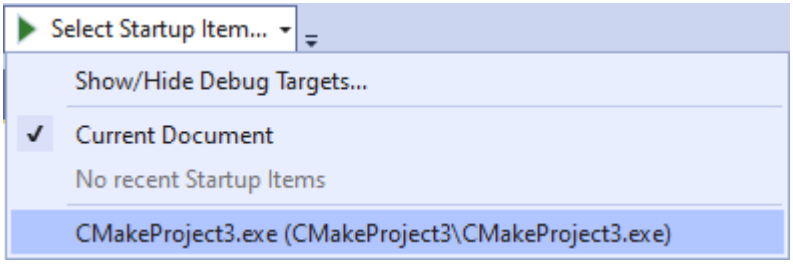
```
    {
      "type": "default",
      "project": "CMakeLists.txt",
      "projectTarget": "CMakeProject3.exe",
      "name": "CMakeProject3.exe"
    }
  ]
}
```

Debug the app

On the host machine, in the Visual Studio **Solution Explorer**, open the CPP file for your CMake project. If you're still in **CMake Targets View**, you'll need to open the **(executable)** node to see it.

The default CPP file is a simple hello world console app. Set a breakpoint on `return 0;`.

On the Visual Studio toolbar, use the **Startup Item** dropdown to select the name you specified for `"name"` in your *`launch.vs.json`* file:

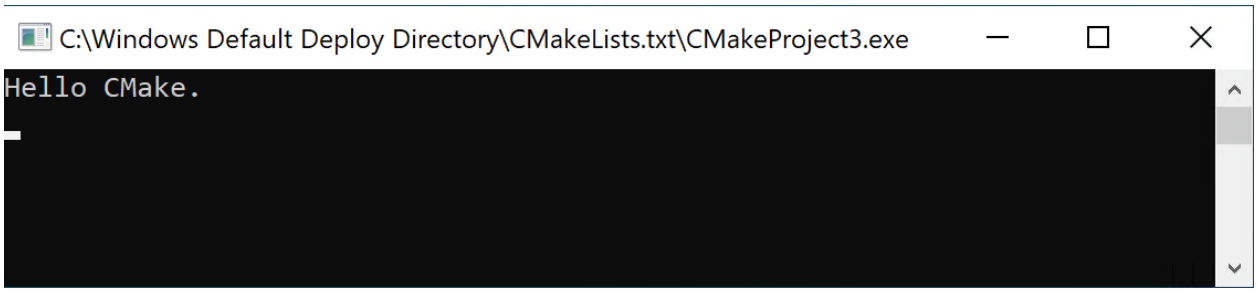


To start debugging, on the Visual Studio toolbar choose **Debug** > **Start Debugging** (or press **F5**).

If it doesn't start, ensure that the following are set correctly in the *`launch.vs.json`* file:

- `"remoteMachineName"` should be set to the IP address, or the machine name, of the remote ARM64 Windows machine.
- `"name"` should match the selection in the Visual Studio startup item dropdown.
- `"projectTarget"` should match the name of the CMake target you want to debug.
- `"type"` should be `"remoteWindows"`
- If the authentication type on the remote debugger is set to **No Authentication**, you should have `"authenticationType": "none"` set in the *`launch.vs.json`* file.
- If you're using Windows authentication, sign in when prompted using an account recognized by the remote machine.

After the project builds, the app should appear on the remote ARM64 Windows machine:



Visual Studio on the host machine should be stopped at the breakpoint for `return 0;`.

What you learned

In this tutorial, you created a CMake project, configured it to build for Windows on ARM64, and debugged it on a remote ARM64 Windows machine.

Next steps

Learn more about configuring and debugging CMake projects in Visual Studio:

- CMake Projects in Visual Studio
- Customize CMake build settings
- Configure CMake debugging sessions
- CMake predefined configuration reference
- launch.vs.jsonschema reference

Clang/LLVM support in Visual Studio CMake projects

Article • 01/09/2023

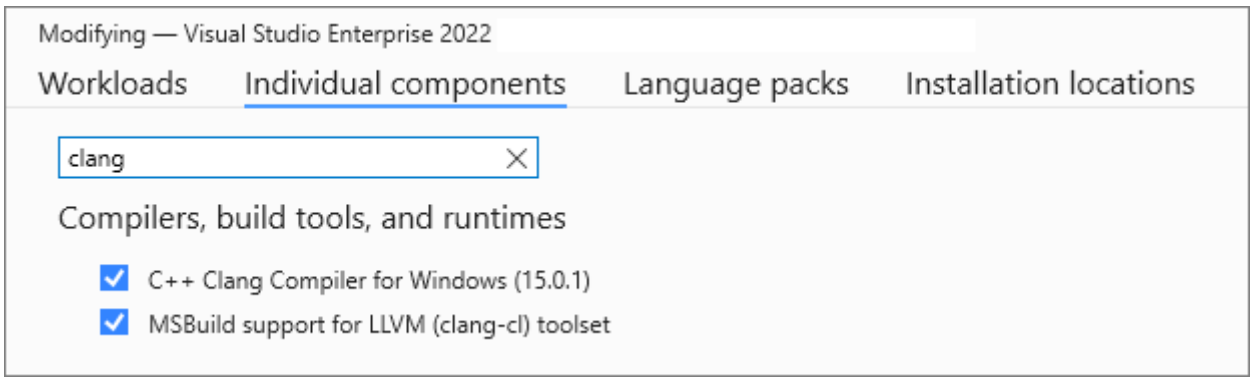
You can use Visual Studio with Clang to edit and debug C++ CMake projects that target Windows or Linux.

Windows: Starting in Visual Studio 2019 version 16.1, Visual Studio includes support for editing, building, and debugging with Clang/LLVM in CMake projects targeting Windows.

Linux: For Linux CMake projects, no special Visual Studio support is required. You can install Clang using your distro's package manager, and add the appropriate commands in the CMakeLists.txt file.

Install

For the best IDE support in Visual Studio, we recommend using the latest Clang compiler tools for Windows. If you don't already have those, you can install them by opening the Visual Studio Installer and choosing **C++ Clang compiler for Windows** under **Desktop development with C++** optional components. You may prefer to use an existing Clang installation on your machine; if so, choose the **MSBuild support for LLVM (clang-cl) toolset** component.

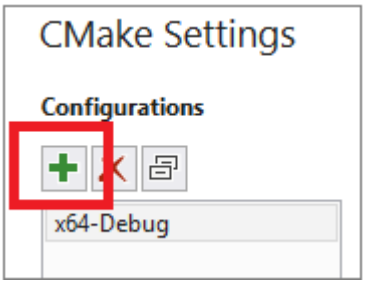


Create a new configuration

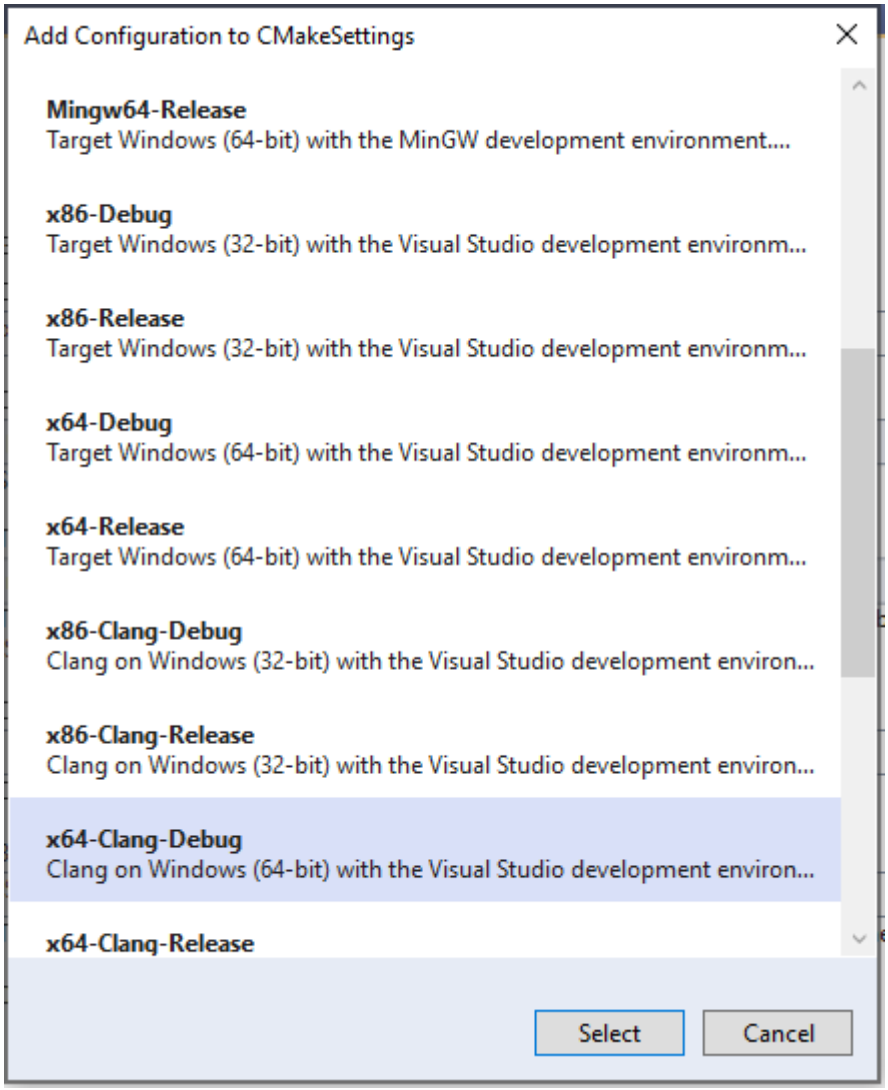
To add a new Clang configuration to a CMake project:

1. Right-click on CMakeLists.txt in **Solution Explorer** and choose **CMake settings for project**.

2. Under **Configurations**, press the **Add Configuration** button:



3. Choose the desired Clang configuration (note that separate Clang configurations are provided for Windows and Linux), then press **Select**:



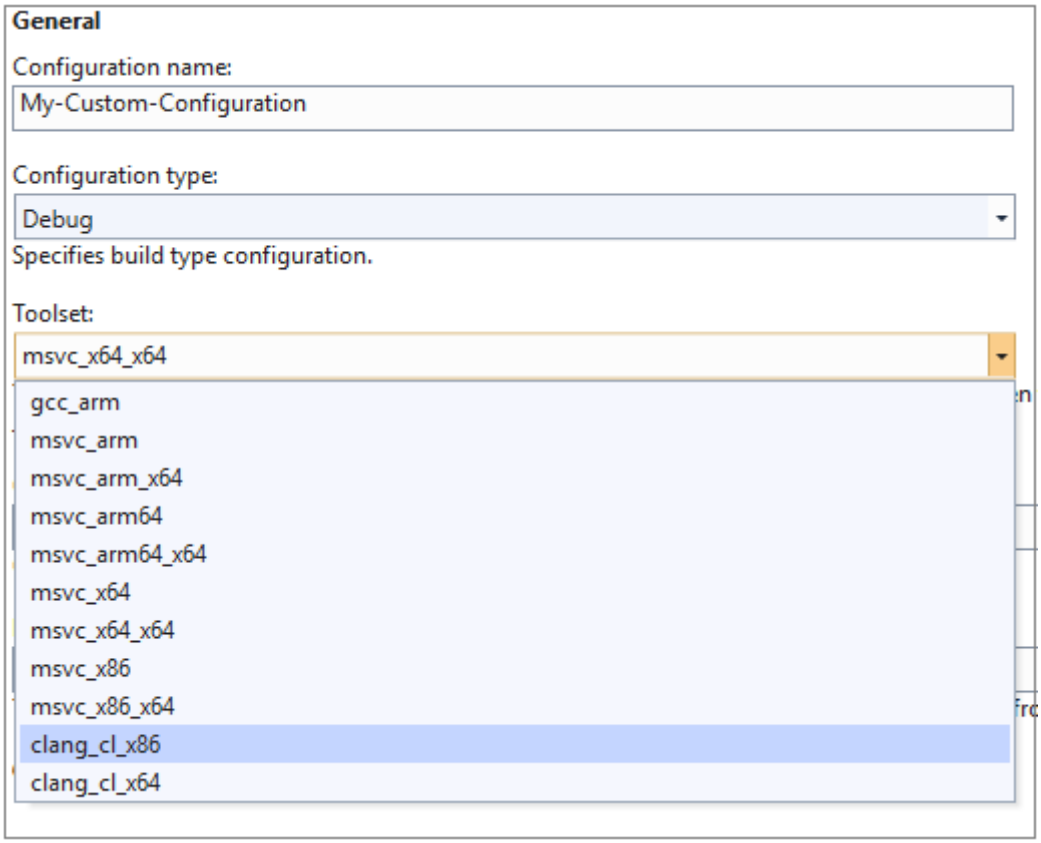
4. To make modifications to this configuration, use the **CMake Settings Editor**. For more information, see Customize CMake build settings in Visual Studio.

Modify an existing configuration to use Clang

To modify an existing configuration to use Clang, follow these steps:

1. Right-click on CMakeLists.txt in **Solution Explorer** and choose **CMake settings for project**.

2. Under **General** select the **Toolset** dropdown and choose the desired Clang toolset:

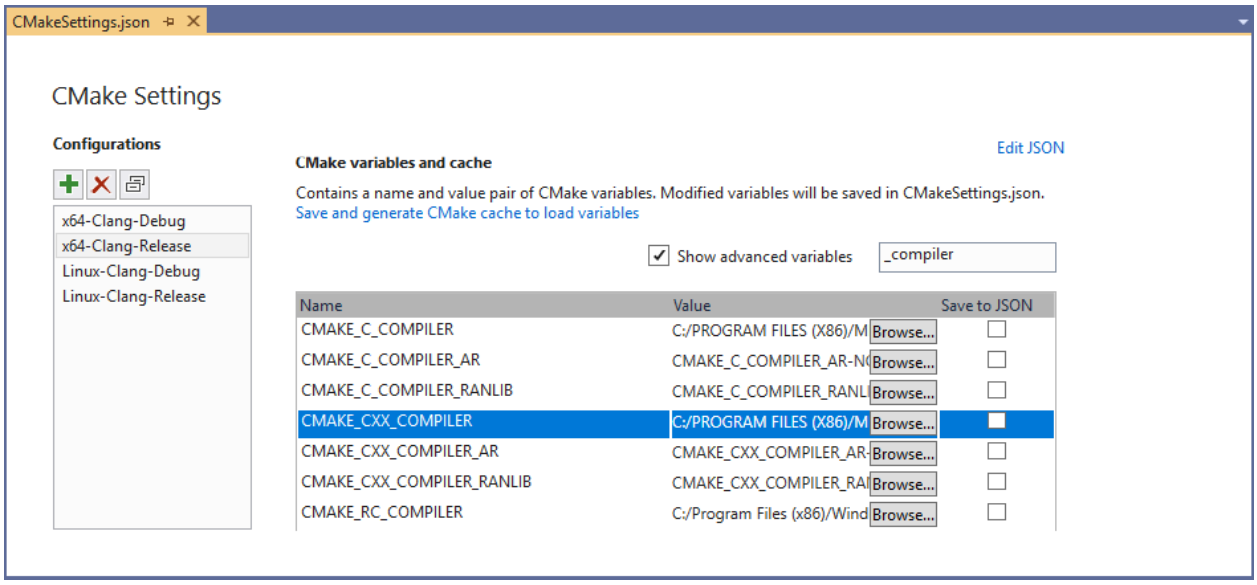


Custom Clang locations

By default, Visual Studio looks for Clang in two places:

- (Windows) The internally installed copy of Clang/LLVM that comes with the Visual Studio installer.
- (Windows and Linux) The PATH environment variable.

You can specify another location by setting the **CMAKE_C_COMPILER** and **CMAKE_CXX_COMPILER** CMake variables in **CMake Settings**:



Clang compatibility modes

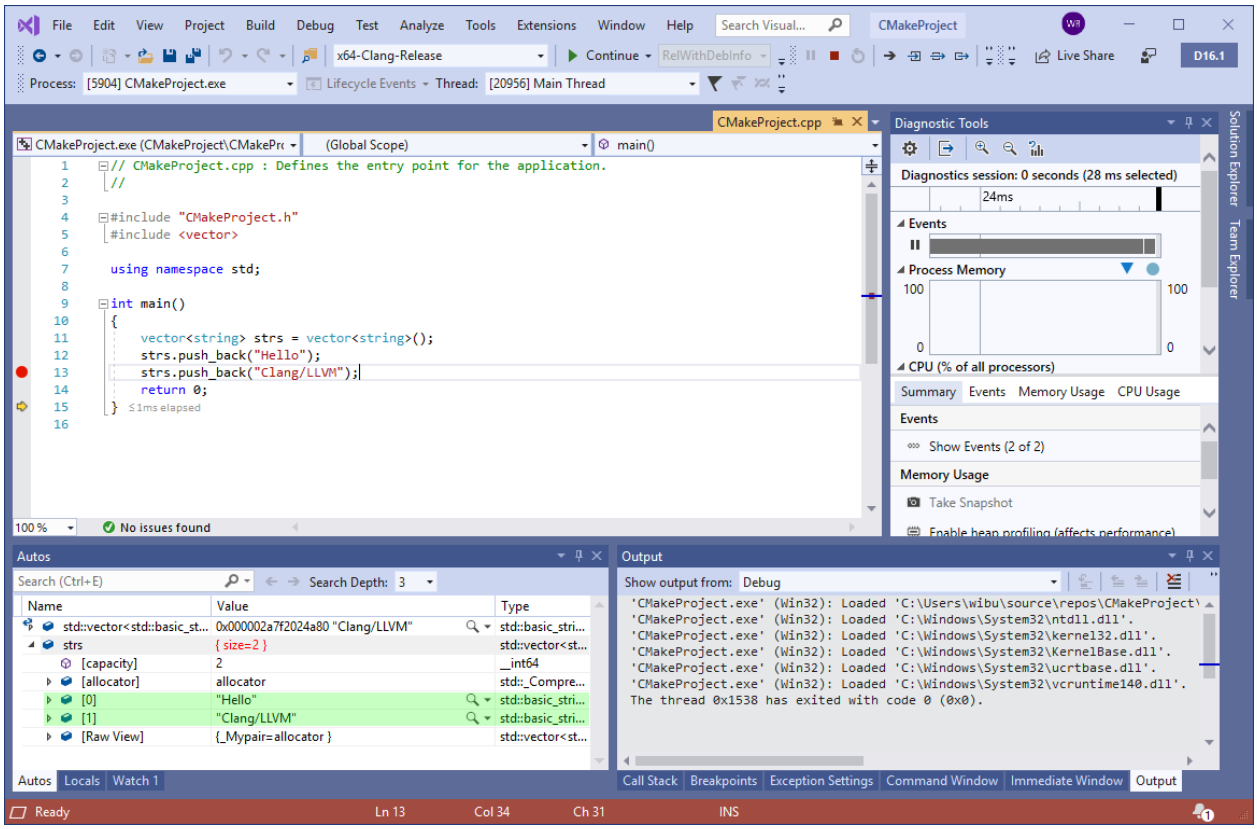
For Windows configurations, CMake by default invokes Clang in clang-cl mode and links with the Microsoft implementation of the Standard Library. By default, **clang-cl.exe** is located in `C:\Program Files (x86)\Microsoft Visual Studio\2019\Common7\IDE\CommonExtensions\Microsoft\Llvm\bin`.

You can modify these values in **CMake Settings** under **CMake variables and cache**. Click **Show advanced variables**. Scroll down to find **CMAKE_CXX_COMPILER**, then click the **Browse** button to specify a different compiler path.

Edit, build, and debug

After you have set up a Clang configuration, you can build and debug the project. Visual Studio detects that you're using the Clang compiler and provides IntelliSense, highlighting, navigation, and other editing features. Errors and warnings are displayed in the **Output Window**.

When debugging, you can use breakpoints, memory and data visualization, and most other debugging features. Some compiler-dependent features such as Edit and Continue aren't available for Clang configurations.


File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
Search Visual...
CMakeProject
x64-Clang-Release
Continue
RelWithDebInfo
Live Share
D16.1
Process: [5904] CMakeProject.exe
Lifecycle Events
Thread: [20956] Main Thread
CMakeProject.cpp
CMakeProject.exe (CMakeProject\CMakePr
(Global Scope)
main()
1 // CMakeProject.cpp : Defines the entry point for the application.
2 //
3
4 #include "CMakeProject.h"
5 #include <vector>
6
7 using namespace std;
8
9 int main()
10 {
11 vector<string> strs = vector<string>();
12 strs.push_back("Hello");
13 strs.push_back("Clang/LLVM");
14 return 0;
15 } ≤1ms elapsed
16
100 %
No issues found
Diagnostic Tools
Diagnostics session: 0 seconds (28 ms selected)
24ms
Events
Process Memory
100
0
100
0
CPU (% of all processors)
Summary Events Memory Usage CPU Usage
Events
Show Events (2 of 2)
Memory Usage
Take Snapshot
Enable heap profiling (affects performance)
Solution Explorer Team Explorer
Autos
Search (Ctrl+E)
Search Depth: 3
Name Value Type
std::vector<std::basic_st... 0x000002a7f2024a80 "Clang/LLVM" std::basic_stri...
strs { size=2 } std::vector<st...
[capacity] 2 __int64
[allocator] allocator std::_Compre...
[0] "Hello" std::basic_stri...
[1] "Clang/LLVM" std::basic_stri...
[Raw View] {_Mypair=allocator } std::vector<st...
Autos Locals Watch 1
Output
Show output from: Debug
'CMakeProject.exe' (Win32): Loaded 'C:\Users\wibu\source\repos\CMakeProject\
'CMakeProject.exe' (Win32): Loaded 'C:\Windows\System32\ntdll.dll'.
'CMakeProject.exe' (Win32): Loaded 'C:\Windows\System32\kernel32.dll'.
'CMakeProject.exe' (Win32): Loaded 'C:\Windows\System32\KernelBase.dll'.
'CMakeProject.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbase.dll'.
'CMakeProject.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140.dll'.
The thread 0x1538 has exited with code 0 (0x0).
Call Stack Breakpoints Exception Settings Command Window Immediate Window Output
Ready
Ln 13
Col 34
Ch 31
INS

Create a CMake Linux project in Visual Studio

Article • 08/03/2021

We recommend you use CMake for projects that are cross-platform or will be made open-source. You can use CMake projects to build and debug the same source code on Windows, the Windows Subsystem for Linux (WSL), and remote systems.

Before you begin

First, make sure you have the Visual Studio Linux workload installed, including the CMake component. That's the **Linux development with C++** workload in the Visual Studio installer. See Install the C++ Linux workload in Visual Studio if you aren't sure you have that installed.

Also, make sure the following are installed on the remote machine:

- gcc
- gdb
- rsync
- zip
- ninja-build (Visual Studio 2019 or above)

You can use Visual Studio 2019 to build and debug on a remote Linux system or WSL, and CMake will be invoked on that system. Cmake version 3.14 or later should be installed on the target machine.

Make sure that the target machine has a recent version of CMake. Often, the version offered by a distribution's default package manager isn't recent enough to support all the features required by Visual Studio. Visual Studio 2019 detects whether a recent version of CMake is installed on the Linux system. If none is found, Visual Studio shows an info-bar at the top of the editor pane. It offers to install CMake for you from https://github.com/Microsoft/CMake/releases .

With Visual Studio 2019, you can create a CMake project from scratch, or open an existing CMake project. To create a new CMake project, follow the instructions below. Or skip ahead to Open a CMake project folder if you already have a CMake project.

Create a new Linux CMake project

To create a new Linux CMake project in Visual Studio 2019:

1. Select **File** > **New Project** in Visual Studio, or press **Ctrl** + **Shift** + **N**.
2. Set the **Language** to **C++** and search for "CMake". Then choose **Next**. Enter a **Name** and **Location**, and choose **Create**.

Alternatively, you can open your own CMake project in Visual Studio 2019. The following section explains how.

Visual Studio creates a minimal *CMakeLists.txt* file with only the name of the executable and the minimum CMake version required. You can manually edit this file however you like; Visual Studio will never overwrite your changes.

To help you make sense of, edit, and author your CMake scripts in Visual Studio 2019, refer to the following resources:

- In-editor documentation for CMake in Visual Studio
- Code navigation for CMake scripts
- Easily Add, Remove, and Rename Files and Targets in CMake Projects

Open a CMake project folder

When you open a folder that contains an existing CMake project, Visual Studio uses variables in the CMake cache to automatically configure IntelliSense and builds. Local configuration and debugging settings get stored in JSON files. You can optionally share these files with others who are using Visual Studio.

Visual Studio doesn't modify the *CMakeLists.txt* files. This allows others working on the same project to continue to use their existing tools. Visual Studio does regenerate the cache when you save edits to *CMakeLists.txt*, or in some cases, to *CMakeSettings.json*. If you're using an **Existing Cache** configuration, then Visual Studio doesn't modify the cache.

For general information about CMake support in Visual Studio, see CMake projects in Visual Studio. Read that before continuing here.

To get started, choose **File** > **Open** > **Folder** from the main menu or else type `devenv.exe <foldername>` in a developer command prompt window. The folder you open should have a *CMakeLists.txt* file in it, along with your source code.

The following example shows a simple *CMakeLists.txt* file and .cpp file:

```
C++
```

```
// hello.cpp

#include <iostream>

int main(int argc, char* argv[])
{
    std::cout << "Hello from Linux CMake \n";
}
```

CMakeLists.txt:

txt

```
cmake_minimum_required(VERSION 3.8)
project (hello-cmake)
add_executable(hello-cmake hello.cpp)
```

Next steps

Configure a Linux CMake project

See also

CMake Projects in Visual Studio

Configure and build with CMake Presets in Visual Studio

Article • 06/09/2023

CMake supports two files that allow users to specify common configure, build, and test options and share them with others: `CMakePresets.json` and `CMakeUserPresets.json`. Use these files to drive CMake in Visual Studio and Visual Studio Code, in a continuous integration (CI) pipeline, and from the command line.

`CMakePresets.json` is for saving project-wide builds. `CMakeUserPresets.json` is for developers to save their own local builds. Both files are supported in Visual Studio 2019 version 16.10 or later.

This article contains information about `CMakePresets.json` integration with Visual Studio. Here are helpful links:

- For more information about the format of `CMakePresets.json`, see the official CMake documentation.
- For more information about the Microsoft vendor maps and macro expansion, see CMakePresets.json and CMakeUserPresets.json Microsoft vendor maps.
- For more information about how to use `CMakePresets.json` in Visual Studio Code, see Configure and build with CMake Presets.

We recommend `CMakePresets.json` as an alternative to `CMakeSettings.json`. Visual Studio never reads from both `CMakePresets.json` and `CMakeSettings.json` at the same time. To enable or disable `CMakePresets.json` integration in Visual Studio, see Enable CMakePresets.json in Visual Studio 2019.

Supported CMake and `CMakePresets.json` versions

The supported `CMakePresets.json` and `CMakeUserPresets.json` schema versions depend on your version of Visual Studio:

- Visual Studio 2019 version 16.10 and later support schema versions 2 and 3.
- Visual Studio 2022 version 17.4 preview 2 adds support for schema versions 4 and 5.

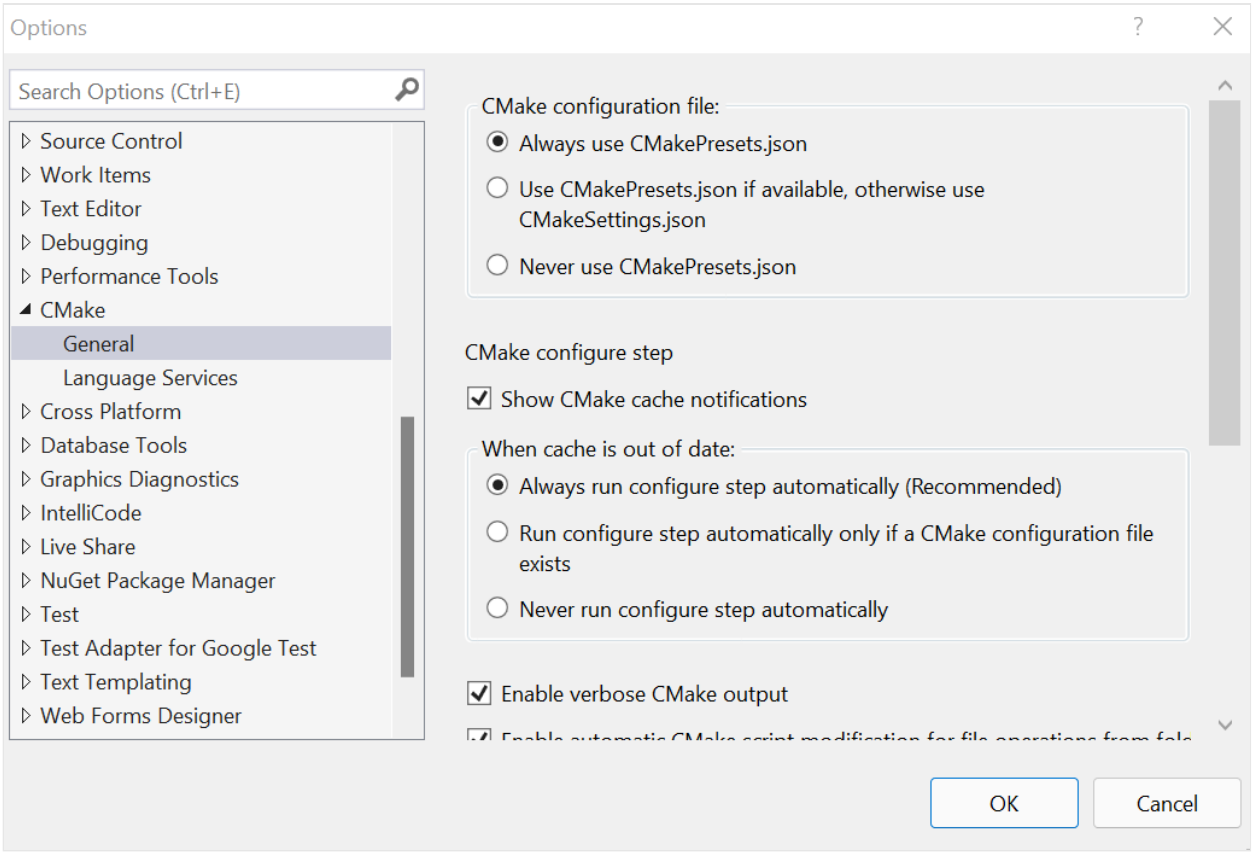
You can update the version by changing the `"version"` field in the root object. For an example and more information, see CMakePresets.json format.

CMake version 3.20 or later is required when you're invoking CMake with `CMakePresets.json` from the command line. However, Visual Studio reads and evaluates `CMakePresets.json` and `CMakeUserPresets.json` itself and doesn't invoke CMake directly with the `--preset` option. So, CMake version 3.20 or later isn't strictly required when you're building with `CMakePresets.json` inside Visual Studio.

We recommend using at least CMake version 3.14 or later.

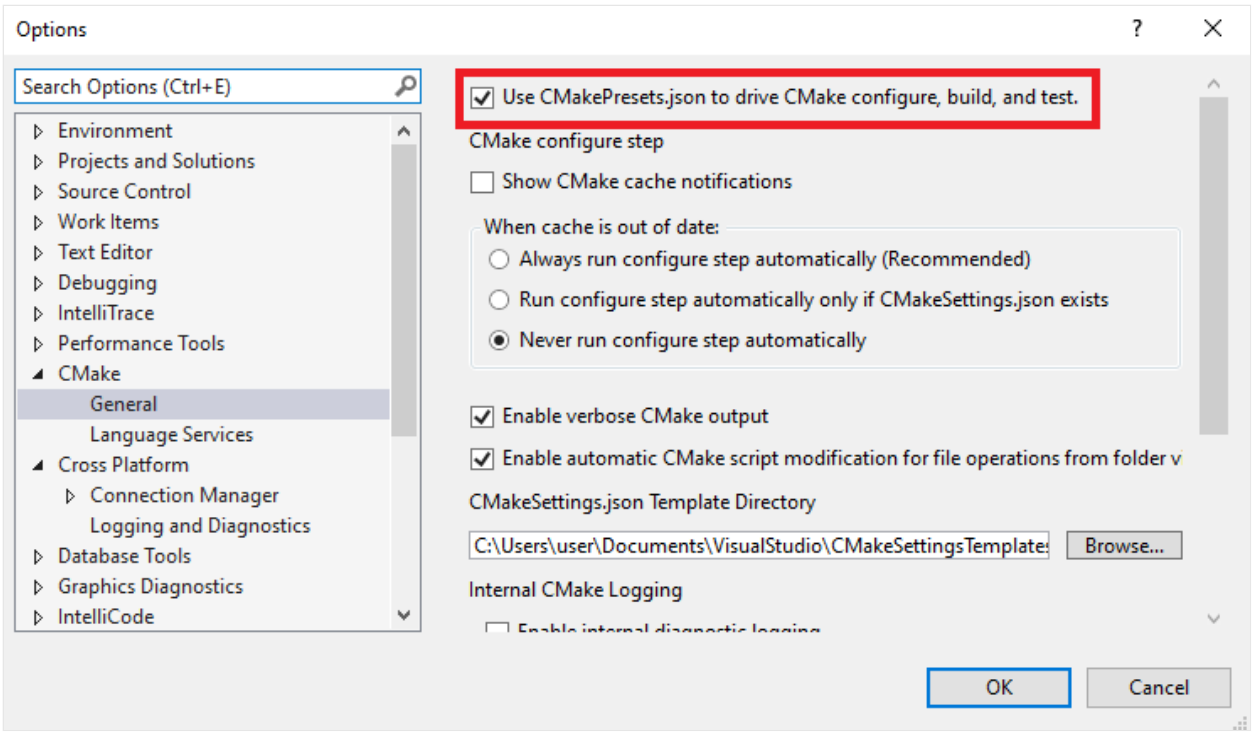
Enable `CMakePresets.json` integration in Visual Studio

`CMakePresets.json` integration isn't enabled by default in Visual Studio. You can enable it in **Tools** > **Options** > **CMake** > **General**:



> **Important**
>
> Close and reopen the folder in Visual Studio to activate the integration.

In some older versions of Visual Studio, **Tools** > **Options** > **CMake** > **General** only has a single option to enable `CMakePresets.json` integration:



The following table indicates when `CMakePresets.json` is used instead of `CMakeSettings.json` to drive CMake configuration and build in Visual Studio 2022 and Visual Studio 2019 version 16.10 and later. If no configuration file is present, default Configure Presets are used.

In the table, "**Tools** > **Options** enabled" means **Use CMakePresets.json to drive CMake configure, build, and test** is selected in **Tools** > **Options** > **CMake** > **General**.

Configuration files	Tools > Options disabled	Tools > Options enabled
No configuration file present	`CMakeSettings.json`	`CMakePresets.json`
`CMakeSettings.json` present	`CMakeSettings.json`	`CMakePresets.json`
`CMakePresets.json` present	`CMakePresets.json`	`CMakePresets.json`
Both configuration files present	`CMakePresets.json`	`CMakePresets.json`

Modify automatic configuration and cache notifications

By default, Visual Studio automatically invokes `configure` each time the active Target System or Configure Preset changes. You can modify this behavior by selecting **Never run configure step automatically** in **Tools** > **Options** > **CMake** > **General**. You can also

disable all CMake cache notifications (gold bars) by clearing **Show CMake cache notifications**.

Default Configure Presets

If no `CMakePresets.json` or `CMakeUserPresets.json` file exists, or if `CMakePresets.json` or `CMakeUserPresets.json` is invalid, Visual Studio falls back on the following default Configure Presets:

Windows example

JSON

```
{
  "name": "windows-default",
  "displayName": "Windows x64 Debug",
  "description": "Sets Ninja generator, compilers, x64 architecture, build
and install directory, debug build type",
  "generator": "Ninja",
  "binaryDir": "${sourceDir}/out/build/${presetName}",
  "architecture": {
    "value": "x64",
    "strategy": "external"
  },
  "cacheVariables": {
    "CMAKE_BUILD_TYPE": "Debug",
    "CMAKE_INSTALL_PREFIX": "${sourceDir}/out/install/${presetName}"
  },
  "vendor": {
    "microsoft.com/VisualStudioSettings/CMake/1.0": {
      "hostOS": [ "Windows" ]
    }
  }
},
```

Linux example

JSON

```
{
  "name": "linux-default",
  "displayName": "Linux Debug",
  "description": "Sets Ninja generator, compilers, build and install
directory, debug build type",
  "generator": "Ninja",
  "binaryDir": "${sourceDir}/out/build/${presetName}",
  "cacheVariables": {
```

```
      "CMAKE_BUILD_TYPE": "Debug",
      "CMAKE_INSTALL_PREFIX": "${sourceDir}/out/install/${presetName}"
    },
    "vendor": {
      "microsoft.com/VisualStudioSettings/CMake/1.0": {
        "hostOS": [ "Linux" ]
      },
      "microsoft.com/VisualStudioRemoteSettings/CMake/1.0": {
        "sourceDir": "$env{HOME}/.vs/$ms{projectDirName}"
     }
    }
  }
```

If you try to open or modify a `CMakePresets.json` file that doesn't exist, Visual Studio automatically creates a `CMakePresets.json` file with the default Configure Presets at the root of your project.

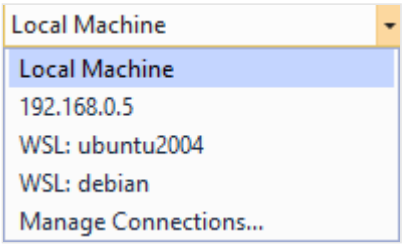
Configure and build

On the Visual Studio toolbar, there are dropdowns for the Target Systems, Configure Presets, and Build Presets when `CMakePresets.json` integration is enabled:



Select a Target System

The dropdown list on the left indicates the active *Target System*. It's the system on which CMake is invoked to configure and build the project. This dropdown list includes your local machine, all SSH connections in Connection Manager by host name, and all Windows Subsystem for Linux (WSL) installations that Visual Studio can find:



In the preceding example:

- **192.168.0.5** is a remote Linux system that was added to Connection Manager.
- **ubuntu2004** and **debian** are WSL installations.

Select **Manage Connections** to open Connection Manager.

Select a Configure Preset

The dropdown list in the middle indicates the active *Configure Preset*. It's the `configurePreset` value that's used when CMake is invoked to generate the project build system. This dropdown list includes the union of non-hidden Configure Presets defined in *`CMakePresets.json`* and *`CMakeUserPresets.json`*.

Visual Studio uses the value of `hostOS` in the Microsoft Visual Studio Settings vendor map to hide Configure Presets that don't apply to the active Target System. For more information, see the entry for `hostOS` in the table under Visual Studio Settings vendor map.

Select **Manage Configurations** to open the *`CMakePresets.json`* file located at the root of the project. *`CMakePresets.json`* is created if it doesn't already exist.

Select a Build Preset

The dropdown list on the right indicates the active *Build Preset*. It's the `buildPreset` value that's used when CMake is invoked to build the project. This dropdown list includes the union of non-hidden Build Presets defined in *`CMakePresets.json`* and *`CMakeUserPresets.json`*.

All Build Presets are required to specify an associated `configurePreset` value. Visual Studio hides Build Presets that don't apply to the active Configure Preset. For more information, see the list of Build Presets.

If there are no Build Presets associated with the active Configure Preset, Visual Studio lists the default Build Preset. The default Build Preset is equivalent to passing `cmake --build` with no other arguments from the command line.

Configure

Visual Studio automatically tries to configure the project when it detects that the CMake cache is out of date. To manually invoke the configuration, select **Project** > **Configure <project-name>** from the main menu. It's the same as running `cmake --preset <configurePreset>` from the command line, where `<configurePreset>` is the name of the active Configure Preset.

To disable automatic cache generation, see Automatic configuration and cache notifications.

Build

To build the entire project, select **Build** > **Build All** from the main menu. It's the same as running `cmake --build --preset <buildPreset>` from the command line, where `<buildPreset>` is the name of the active Build Preset.

To build a single target, switch to **CMake Targets View** in Solution Explorer. Then right-click any target and select **Build** from the shortcut menu.

> ⓘ **Note**
>
> Visual Studio 2019 doesn't support the `buildPresets.targets` option to build a subset of targets specified in *`CMakePresets.json`*.

Run CTest

`CMakePresets.json` supports two menu options in Visual Studio 2019:

- **Test** > **Run CTests** for <project-name> invokes CTest and runs all tests associated with the active Configure Preset and Build Preset, with no other arguments passed to CTest.
- **Test** > **Run Test Preset** for <configurePreset> expands to show all Test Presets associated with the active Configure Preset. Selecting a single Test Preset is the same as running `ctest --preset <testPreset>` from the command line, where `<testPreset>` is the name of the selected Test Preset. This option is unavailable if no Test Presets are defined for the active Configure Preset.

In Visual Studio 2019, Test Explorer isn't integrated with *`CMakePresets.json`*.

Add new presets

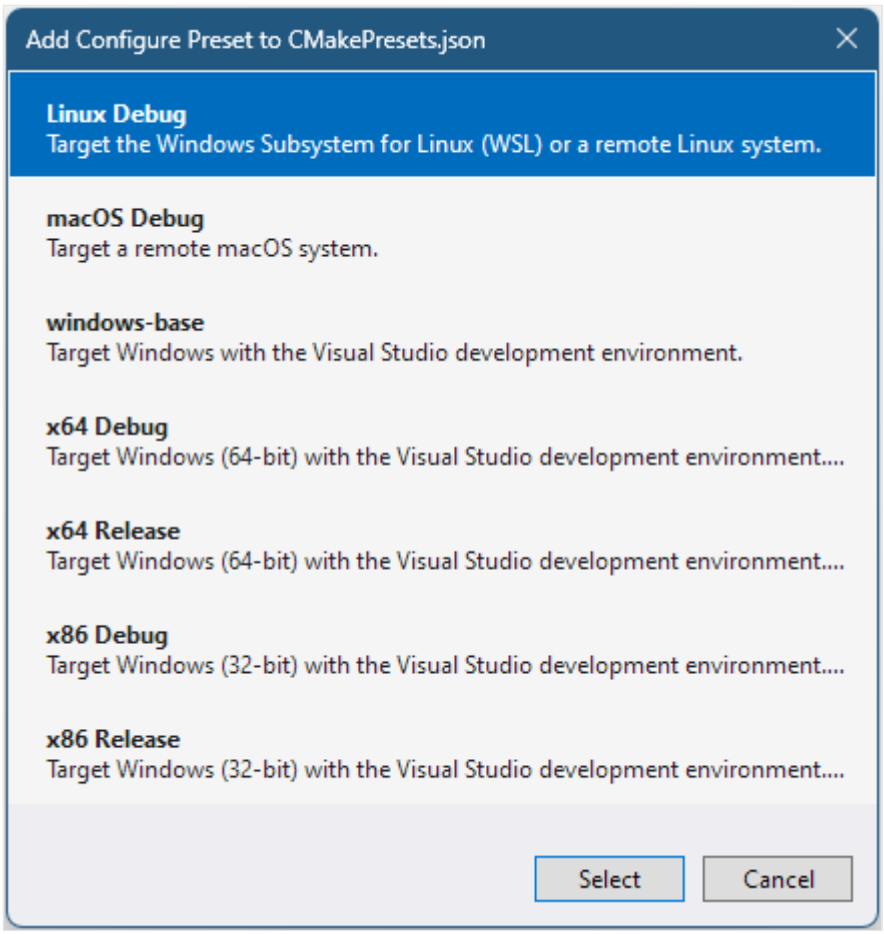
In Visual Studio 2019, all commands and preset templates modify *`CMakePresets.json`*. You can add new user-level presets by directly editing *`CMakeUserPresets.json`*.

Use a forward slash (`/`) for paths in *`CMakePresets.json`* and *`CMakeUserPresets.json`*.

Add new Configure Presets

To add a new Configure Preset to *`CMakePresets.json`*, from **Solution Explorer**, right-click *`CMakePresets.json`* from **Folder View** and select **Add Configuration** from the shortcut

menu. The dialog to select a Configure Preset template appears:



Select the **Windows x64 Debug** template to configure on Windows systems. Select the **Linux Debug** template to configure on WSL and remote Linux systems. For more information about editing `CMakePresets.json`, see Edit presets.

The selected template is added to `CMakePresets.json` if it exists. Otherwise, the template is copied into a new `CMakePresets.json` file.

Add new Build Presets and Test Presets

Visual Studio 2019 doesn't offer templates for new Build Presets and Test Presets. You can add Build Presets and Test Presets by directly editing `CMakePresets.json`. For more information, see the list of Build Presets, the list of Test Presets, or an example CMakePresets.json file.

Edit presets

The official CMake documentation is the best resource for editing Configure Presets, Build Presets, and Test Presets. The following information is a subset of the CMake documentation that's especially relevant to Visual Studio developers.

Select your compilers

You can set C and C++ compilers by using `cacheVariables.CMAKE_C_COMPILER` and `cacheVariables.CMAKE_CXX_COMPILER` in a Configure Preset. It's equivalent to passing `-D CMAKE_C_COMPILER=<value>` and `-D CMAKE_CXX_COMPILER=<value>` to CMake from the command line. For more information, see CMAKE_<LANG>_COMPILER.

Use the following examples to build with `cl.exe` and `clang-cl.exe` from Visual Studio. The C++ Clang tools for Windows components must be installed for you to build with `clang-cl`.

Build with `cl.exe`:

JSON

```
"cacheVariables": {
  "CMAKE_BUILD_TYPE": "Debug",
  "CMAKE_INSTALL_PREFIX": "${sourceDir}/out/install/${presetName}",
  "CMAKE_C_COMPILER": "cl",
  "CMAKE_CXX_COMPILER": "cl"
},
```

Build with `clang`:

JSON

```
"cacheVariables": {
  "CMAKE_BUILD_TYPE": "Debug",
  "CMAKE_INSTALL_PREFIX": "${sourceDir}/out/install/${presetName}",
  "CMAKE_C_COMPILER": "clang-cl",
  "CMAKE_CXX_COMPILER": "clang-cl"
},

"vendor": {
  "microsoft.com/VisualStudioSettings/CMake/1.0": {
    "intelliSenseMode": "windows-clang-x64"
  }
}
```

If you use either `Visual Studio 16 2019` or `Visual Studio 17 2022` as your generator, you can use the `toolset` Configure Preset to specify the `ClangCL` toolset:

JSON

```
"cacheVariables": {
  "CMAKE_BUILD_TYPE": "Debug",
  "CMAKE_INSTALL_PREFIX": "${sourceDir}/out/install/${presetName}",
```

```
},

"toolset": "ClangCL",

"vendor": {
  "microsoft.com/VisualStudioSettings/CMake/1.0": {
    "intelliSenseMode": "windows-clang-x64"
  }
}
```

For more information on generators that support the `toolset` specification, see CMAKE_GENERATOR_TOOLSET in the CMake documentation.

> **Important**
>
> In Visual Studio 2019, you must explicitly specify a Clang IntelliSense mode when you're building with `clang` or `clang-cl`.

To reproduce these builds outside Visual Studio, see Run CMake from the command line or a CI pipeline.

To build on Linux or without the Visual C++ toolset, specify the name of a compiler on your `PATH` instance, or an environment variable that evaluates to the full path of a compiler. Full paths are discouraged so that the file can remain shareable. A preset that builds with GCC version 8 might look like this:

JSON

```
"cacheVariables": {
  "CMAKE_BUILD_TYPE": "Debug",
  "CMAKE_INSTALL_PREFIX": "${sourceDir}/out/install/${presetName}",
  "CMAKE_C_COMPILER": "gcc-8",
  "CMAKE_CXX_COMPILER": "g++-8"
},
```

You can also set compilers with a CMake toolchain file. Toolchain files can be set with `cacheVariables.CMAKE_TOOLCHAIN_FILE`, which is equivalent to passing `-D CMAKE_TOOLCHAIN_FILE=<value>` to CMake from the command line. A CMake toolchain file is most often used for cross-compilation. For more information about authoring CMake toolchain files, see CMake toolchains.

Select your generator

The Windows and Linux Configure Preset templates both specify Ninja as the default generator. Other common generators are the Visual Studio Generators on Windows and Unix Makefiles on Linux and macOS. You can specify a new generator with the `generator` option in a Configure Preset. It's equivalent to passing `-G` to CMake from the command line.

Set `architecture.strategy` and `toolset.strategy` to `set` when you're building with a Visual Studio Generator. For more information, see CMake generators.

Select your configuration type

You can set the configuration type (`Debug` or `Release`) for single configuration generators by using `cacheVariables.CMAKE_BUILD_TYPE`. It's equivalent to passing `-D CMAKE_BUILD_TYPE=<value>` to CMake from the command line. For more information, see CMAKE_BUILD_TYPE.

Select your target and host architecture when building with the Visual C++ toolset

You can set the target architecture (x64, Win32, ARM64, or ARM) by using `architecture.value`. It's equivalent to passing `-A` to CMake from the command line. For more information, see Platform Selection.

> **Note**
>
> Currently, Visual Studio Generators expect the Win32 syntax and command-line generators (like Ninja) expect the x86 syntax when you're building for x86.

You can set the host architecture (x64 or x86) and toolset by using `toolset.value`. It's equivalent to passing `-T` to CMake from the command line. For more information, see Toolset Selection.

The `architecture.strategy` and `toolset.strategy` values tell CMake how to handle the architecture and toolset fields. `set` means CMake sets the respective value, and `external` means CMake won't set the respective value.

We recommend using `set` with IDE generators like the Visual Studio Generator. Use `external` with command-line generators like Ninja. These values allow vendors like Visual Studio to supply the required environment before CMake is invoked. For more information about the architecture and toolset fields, see the list of Configure Presets.

If you don't want to source an environment, you can set `architecture.strategy` to `external` and `architecture.value` to `unspecified`. You might find it useful not to source an environment for any one of these reasons:

- You use a toolset other than MSVC.
- You use a custom toolchain, such as in embedded scenarios.
- You don't need a specific environment to build.

For a full list of IDE generators that support the architecture field, see CMAKE_GENERATOR_PLATFORM. For a full list of IDE generators that support the toolset field, see CMAKE_GENERATOR_TOOLSET.

Use the following examples to target ARM64 with the Ninja generator, or to target Win32 (x86) with the Visual Studio 16 2019 generator:

JSON

```
"generator": "Ninja",
"architecture": {
    "strategy": "external",
    "value": "arm64"
},

"generator": "Visual Studio 16 2019",
"architecture": {
    "strategy": "set",
     "value": "Win32"
},
```

Set and reference environment variables

You can set environment variables by using the environment map. Environment variables are inherited through the `inherits` field, but you can override them as you like.

A preset's environment is the union of its own environment and the environment from all its parents. If multiple `inherits` presets provide conflicting values for the same variable, the earlier preset in the `inherits` list is preferred. You can unset a variable inherited from another preset by setting it to `null`.

Environment variables set in a Configure Preset also automatically flow through to associated Build Presets and Test Presets, unless `inheritConfigureEnvironment` is set to `false`. For more information, see the list of Configure Presets.

You can reference environment variables by using the `$env{<variable-name>}` and `$penv{<variable-name>}` syntax. For more information, see Macro Expansion.

Configure IntelliSense for a cross-compiler

By default, Visual Studio uses the IntelliSense mode that matches your specified toolset and target architecture. If you're cross-compiling, you might need to manually specify the correct IntelliSense mode by using the `intelliSenseMode` option in the Visual Studio Settings vendor map. For more information, see the entry for `intelliSenseMode` in the table under Visual Studio Settings vendor map.

Configure and build on a remote system or the Windows Subsystem for Linux

With *`CMakePresets.json`* support in Visual Studio, you can easily configure and build your project on Windows, WSL, and remote systems. The steps to configure and build your project on Windows, a remote system, or WSL are the same. However, a few behaviors are specific to remote development.

`${sourceDir}` behavior in remote copy scenarios

In local scenarios (including WSL1), `${sourceDir}` evaluates to the path to the project source directory that's open in Visual Studio. In remote copy scenarios, `${sourceDir}` evaluates to the path to the project source directory on the Target System and not the project source directory on the local machine.

The value of `sourceDir` in the Visual Studio Remote Settings vendor map determines the project source directory on the Target System (defaults to `$env{HOME}/.vs/$ms{projectDirName}`). For more information, see the entry for `sourceDir` in the table under Visual Studio Settings vendor map.

Local folder for remote output

Remote copy scenarios require a local directory to copy some remote files like CMake File API response files or build files if `copyBuildOutput` in the Visual Studio Remote Settings vendor map is set to `true`. These files are automatically copied to `<local-source-directory>/out/<remote-connection-ID>/build/${presetName}`.

Invoking the same Configure Preset on Windows and WSL1

You'll see an error if you try to use the same Configure Preset on Windows and WSL1. Windows and WSL1 both use the Windows file system, so CMake will try to use the same output directory (`binaryDir`) for both the Windows and WSL1 build trees.

If you want to use the same Configure Preset with both Windows and the WSL1 toolset, create a second Configure Preset that inherits from the original preset and specifies a new `binaryDir` value. In the following example, `windows-preset` can be used on Windows and `base-preset` can be used on WSL1:

JSON

```
{
  "name": "windows-preset",
  "inherits": "base-preset",
  "binaryDir": "${sourceDir}/out/build/${presetName}",
  "vendor": {
    "microsoft.com/VisualStudioSettings/CMake/1.0": {
      "hostOS": "Windows"
    }
  }
}
```

> **Note**
>
> In Visual Studio 2019, only the WSL1 toolset is supported. You'll see this behavior any time you invoke `configure` on both Windows and WSL.

Enable vcpkg integration

Vcpkg helps you manage C and C++ libraries on Windows, Linux, and macOS. A vcpkg toolchain file (`vcpkg.cmake`) must be passed to CMake to enable vcpkg integration. For more information, see the vcpkg documentation.

Visual Studio no longer passes your vcpkg toolchain file to CMake automatically when *`CMakePresets.json`* integration is enabled. This change eliminates Visual Studio-specific behavior and ensures that you can reproduce your build from the command line.

Instead, set the path to `vcpkg.cmake` by using the `VCPKG_ROOT` environment variable in *`CMakePresets.json`*:

JSON

```
"cacheVariables": {
   "CMAKE_TOOLCHAIN_FILE": {
      "value": "$env{VCPKG_ROOT}/scripts/buildsystems/vcpkg.cmake",
       "type": "FILEPATH"
    }
 },
```

`VCPKG_ROOT` should be set to the root of your vcpkg installation. For more information, see vcpkg environment variables.

If you're already using a CMake toolchain file and want to enable vcpkg integration, see Using multiple toolchain files. Follow those instructions to use an external toolchain file with a project by using vcpkg.

Variable substitution in *launch.vs.json* and *tasks.vs.json*

CMakePresets.json supports variable substitution in *launch.vs.json* and *tasks.vs.json*. Here are some considerations:

- Environment variables set in the active Configure Preset automatically flow through to *launch.vs.json* and *tasks.vs.json* configurations. You can unset individual environment variables in *launch.vs.json* and *tasks.vs.json* by setting them to `null`. The following example sets the variable `DEBUG_LOGGING_LEVEL` to `null` in *launch.vs.json*: `"env": { "DEBUG_LOGGING_LEVEL": null }`.
- Key values set in the active Configure Preset are available for consumption in *launch.vs.json* and *tasks.vs.json* with the syntax `${cmake.<KEY-NAME>}`. For example, use `${cmake.binaryDir}` to reference the output directory of the active Configure Preset.
- Individual environment variables set in the environment map of the active Configure Preset are available for consumption in *launch.vs.json* and *tasks.vs.json* through the syntax `${env.<VARIABLE-NAME>}`.

Update your *launch.vs.json* and *task.vs.json* files to reference *CMakePresets.json* syntax instead of *CMakeSettings.json* syntax. Macros that reference the old *CMakeSettings.json* syntax when *CMakePresets.json* is the active configuration file are slated for deprecation in a future release. For example, reference the output directory of

the active Configure Preset with `${cmake.binaryDir}` instead of `${cmake.buildRoot}`, because *`CMakePresets.json`* uses the `binaryDir` syntax.

Troubleshoot

If things aren't working as expected, you can try a few troubleshooting steps.

If either *`CMakePresets.json`* or *`CMakeUserPresets.json`* is invalid, Visual Studio will fall back on its default behavior and show only the default Configure Presets. Visual Studio IntelliSense can help you catch many of these JSON errors, but it won't know if you're referencing a preset with `inherits` or `configurePreset` by the wrong name.

To check if your preset files are valid, run `cmake --list-presets` from the command line at the root of your project directory. (CMake 3.20 or later is required.) If either file is invalid, you'll see the following error:

Windows Command Prompt

```
CMake Error: Could not read presets from
C:/Users/<user>/source/repos/<project-name>: JSON parse error
```

Other troubleshooting steps include:

- Delete the cache and reconfigure the project (**CMake: Delete Cache** and **Project** > **Configure <project-name>**).
- Close and reopen the folder in Visual Studio (**File** > **Close Folder**).
- Delete the `.vs` folder at the root your project.

If you've identified a problem, the best way to report it is by selecting the **Send Feedback** button in the upper-right corner of Visual Studio.

Enable logging for remote connections

You can enable logging for remote connections if you're having trouble connecting or copying files to a remote system. For more information, see Logging for remote connections.

Enable AddressSanitizer for Windows and Linux

Visual Studio supports AddressSanitizer (ASAN), a C and C++ runtime memory error detector, for both Windows and Linux development. The `addressSanitizerEnabled`

option in `CMakeSettings.json` enables AddressSanitizer. `CMakePresets.json` doesn't support this behavior.

Instead, enable and disable AddressSanitizer by setting the required compiler and linker flags yourself. Setting them removes Visual Studio-specific behavior and ensures that the same `CMakePresets.json` file can reproduce your build from the command line.

You can add the following sample to `CMakeLists.txt` to enable or disable AddressSanitizer for a target:

Windows Command Prompt

```
option(ASAN_ENABLED "Build this target with AddressSanitizer" ON)

if(ASAN_ENABLED)
  if(MSVC)
    target_compile_options(<target> PUBLIC /fsanitize=address)
  else()
    target_compile_options(<target> PUBLIC -fsanitize=address <additional-
options>)
    target_link_options(<target> PUBLIC -fsanitize=address)
  endif()
endif()
```

The `<additional-options>` part lists other compilation flags, like `"-fno-omit-frame-pointer"`. For more information about AddressSanitizer for Linux, see Using AddressSanitizer. For more information about using AddressSanitizer with MSVC, see Use AddressSanitizer from a developer command prompt.

Pass runtime flags to AddressSanitizer by using the `ASAN_OPTIONS` field in `launch.vs.json`. `ASAN_OPTIONS` defaults to `detect_leaks=0` when no other runtime options are specified because LeakSanitizer isn't supported in Visual Studio.

Run CMake from the command line or a CI pipeline

You can use the same `CMakePresets.json` and `CMakeUserPresets.json` files to invoke CMake in Visual Studio and from the command line. The CMake and CTest documentation are the best resources for invoking CMake and CTest with `--preset`. CMake version 3.20 or later is required.

Sourcing the environment when building with command-line generators on Windows

It's up to the user to configure the environment before CMake is invoked in building with a command-line generator. If you're building with Ninja and the Visual C++ toolset on Windows, set the environment before CMake is called to generate the build system. You can do it by calling `vcvarsall.bat` with the `architecture` argument. The `architecture` argument specifies the host and target architecture to use. For more information, see vcvarsall syntax. If you build on Linux or on Windows with a Visual Studio Generator, you don't need to take this step.

It's the same step that Visual Studio takes for you when the IDE invokes CMake. Visual Studio parses the active Configure Preset for the host and target architecture specified by `toolset` and `architecture`. Visual Studio then sources the specified environment from `vcvarsall.bat`. When you build from the Windows command line with Ninja, you'll need to take this step yourself.

`vcvarsall.bat` is installed with the Build Tools for Visual Studio. By default, `vcvarsall.bat` is installed in `C:\Program Files (x86)\Microsoft Visual Studio\2019\<edition>\VC\Auxiliary\Build`. You can add `vcvarsall.bat` to `PATH` if you use the command-line workflow often.

Example command-line workflow

You can use the following commands to configure and build a CMake project that uses Ninja to target ARM64 with x64 build tools. CMake version 3.20 or later is required. Run these commands from the directory where your `CMakePresets.json` file is located:

Windows Command Prompt

```
/path/to/vcvarsall.bat x64_arm64
cmake --list-presets=all .
cmake --preset <configurePreset-name>
cmake --build --preset <buildPreset-name>
```

Example `CMakePresets.json` file

The `CMakePresets.json` file in box2d-lite contains examples of Configure Presets, Build Presets, and Test Presets. For more information about this example, see the presentation An Introduction to CMakePresets.json. You can see another example in the DirectXTK project, which shows many build targets in its `configurePresets` section.

Next steps

Learn more about configuring and debugging CMake projects in Visual Studio:

- CMake Projects in Visual Studio
- Customize CMake build settings
- Configure CMake debugging sessions
- CMake predefined configuration reference

`CMakePresets.json` and `CMakeUserPresets.json` Microsoft vendor maps

Article • 02/08/2022

CMake supports two files, `CMakePresets.json` and `CMakeUserPresets.json`, that allow users to specify common configure, build, and test options and share them with others.

`CMakePresets.json` and `CMakeUserPresets.json` can be used to drive CMake in Visual Studio, in Visual Studio Code, in a Continuous Integration (CI) pipeline, and from the command line.

`CMakePresets.json` is intended to save project-wide builds, and `CMakeUserPresets.json` is intended for developers to save their own local builds. The schema for both files is identical.

`CMakePresets.json` and `CMakeUserPresets.json` support vendor maps to store vendor-specific information. Microsoft maintains two vendor maps with options specific to Visual Studio and Visual Studio Code. Here we document two Microsoft vendor maps and vendor macros. For more information about the rest of the schema, see the official CMake documentation. It includes information about Configure Presets, Build Presets, and Test Presets.

For more information about how to use `CMakePresets.json` in Visual Studio, see Configure and build with CMake Presets in Visual Studio

For more information about how to use `CMakePresets.json` in Visual Studio Code, see Configure and build with CMake Presets in VS Code

Visual Studio Settings vendor map

One vendor map with the vendor URI `microsoft.com/VisualStudioSettings/CMake/<version>` is allowed per Configure Preset and contains options specific to CMake integration in Visual Studio and Visual Studio Code. All options in the vendor map apply to Visual Studio. Options that apply to both Visual Studio and Visual Studio Code have been explicitly marked.

All settings in the Visual Studio Settings vendor map are optional and inherited from Configure Presets specified by the `inherits` key. Only options that have been modified

are written to the file. The Visual Studio Settings vendor map is supported by both `CMakePresets.json` and `CMakeUserPresets.json`.

The options in the Visual Studio Settings vendor map don't affect the construction of the CMake or CTest command line. It's so the same `CMakePresets.json` file can be used to drive CMake with Visual Studio, Visual Studio Code, and from the command line. The exceptions are the `cacheRoot` and `cmakeGenerateCommand` options. These options are specific to the Open Existing Cache scenario in Visual Studio and can't be reproduced from the command line.

Setting	Description
`hostOS`	An array of supported operating systems (OS). Accepted values are `Windows`, `Linux`, and `macOS`. The value of `hostOS` is used by Visual Studio and Visual Studio Code to hide Configure Presets that don't apply to the OS of the target system and provide a better user experience. If `hostOS` is unspecified, then Visual Studio and Visual Studio Code will always show all Configure Presets for selection. This field can also be a string, which is equivalent to an array containing one string This option is supported by both Visual Studio and Visual Studio Code.

Setting	Description
`intelliSenseMode`	Specifies the mode used for computing IntelliSense information in Visual Studio with the format `<target>-<toolset>-<arch>`. Accepted values: `android-clang-arm` `android-clang-arm64` `android-clang-x6` `android-clang-x86` `ios-clang-ar` `ios-clang-arm64` `ios-clang-x6` `ios-clang-x86` `linux-gcc-arm` `linux-gcc-x64` `linux-gcc-x86` `windows-clang-arm` `windows-clang-arm64` `windows-clang-x64` `windows-clang-x86` `windows-msvc-arm` `windows-msvc-arm64` `windows-msvc-x64` `windows-msvc-x86` If `intelliSenseMode` is unspecified, then Visual Studio uses the IntelliSense mode that matches your specified compilers and target architecture. `intelliSenseMode` is often used to provide improved IntelliSense for cross-compilation. In Visual Studio 2019, you must explicitly specify a clang IntelliSense mode when building with clang or clang-cl.
`intelliSenseOptions`	A map of extra IntelliSense configuration options. `useCompilerDefaults`: A `bool` that specifies whether to use the compiler default defines and include paths for IntelliSense. Should only be `false` if the compilers in use don't support gcc-style arguments. Defaults to `true`. `additionalCompilerArgs`: An array of extra options to control IntelliSense in Visual Studio. This option supports macro expansion.
`enableMicrosoftCodeAnalysis`	A `bool` that enables Microsoft code analysis in Visual Studio when building with `cl` or `clang-cl`. Defaults to `false`.

Setting	Description
`codeAnalysisRuleset`	Specifies the ruleset to use when running Microsoft code analysis in Visual Studio. You can use a path to a ruleset file, or the name of a ruleset file installed with Visual Studio. This option supports macro expansion.
`disableExternalAnalysis`	A `bool` that specifies whether code analysis should run on external headers in Visual Studio.
`codeAnalysisExternalRuleset`	Specifies the ruleset to use when running Microsoft code analysis on external header in Visual Studio. You can use a path to a ruleset file, or the name of a ruleset file installed with Visual Studio. This option supports macro expansion.
`enableClangTidyCodeAnalysis`	A bool that enables clang-tidy code analysis in Visual Studio when building with `clang-cl`. Defaults to `false`.
`clangTidyChecks`	A comma-separated list of warnings passed to clang-tidy when running clang-tidy code analysis in Visual Studio. Wildcards are allowed, and the `-` prefix will remove checks.
`cacheRoot`	Specifies the path to a CMake cache. This directory should contain an existing *`CMakeCache.txt`* file. This key is only supported by the Open Existing Cache scenario in Visual Studio. This option supports macro expansion.
`cmakeGenerateCommand`	A command-line tool (specified as a command-line program and arguments, for example, **`gencache.bat debug`**) to generate the CMake cache. This command runs in the shell using the specified environment of the preset when CMake configure is invoked. This key is only supported by the Open Existing Cache scenario in Visual Studio. This option supports macro expansion.

Visual Studio Remote Settings vendor map

One vendor map with the vendor URI `microsoft.com/VisualStudioRemoteSettings/CMake/<version>` is allowed per Configure Preset and contains options specific to remote development in Visual Studio. Remote development means you're invoking CMake on a remote SSH connection or WSL. None of the options in the Visual Studio Remote Settings vendor map apply to Visual Studio Code.

All settings in the Visual Studio Remote Settings vendor map are optional and inherited from Configure Presets specified by the `inherits` key. Only options that have been

modified are written to the file. The Visual Studio Remote Settings vendor map is supported by both *`CMakePresets.json`* and *`CMakeUserPresets.json`*.

The options in the Visual Studio Settings vendor map don't affect the construction of the CMake or CTest command line. It's so the same *`CMakePresets.json`* file can be used to drive CMake with Visual Studio, Visual Studio Code, and from the command line.

Many of the options in the Visual Studio Remote Settings vendor map are ignored when targeting WSL1. It's because the WSL1 toolset executes all commands locally and relies on Windows drives mounted under the `/mnt` folder to access local source files from WSL1. No source file copy is required. Options that are ignored when targeting WSL1 have been explicitly marked.

Setting	Description
`sourceDir`	Path to the directory on the remote system where the project will be copied. Defaults to `$env{HOME}/.vs/$ms{projectDirName}`. This option supports macro expansion. In remote copy scenarios, the macro `${sourceDir}` evaluates to the project source directory on the remote system and not the project source directory on the Windows machine. Remote copy scenarios include targeting a remote SSH connection. In these cases, the project source directory on the remote system is determined by the value of `sourceDir` in the Visual Studio Remote Settings vendor map. This option is ignored when targeting WSL1.
`copySources`	If `true`, Visual Studio will copy sources from Windows to the remote system. Set to `false` if you manage file synchronization yourself. Defaults to `true`. This option is ignored when targeting WSL1.

Setting	Description
`copySourcesOptions`	An object of options related to the source copy from Windows to the remote system. This object is ignored when targeting WSL1. `copySourcesOptions.exclusionList`: A list of paths to be excluded when copying source files to the remote system. A path can be the name of a file or directory, or a relative path from the root of the copy. Defaults to `[ ".vs", ".git", "out" ]`. This option supports macro expansion. `copySourcesOptions.method`: The method used to copy source files to the remote system. Accepted values are `rsync` and `sftp`. Defaults to `rsync`. `copySourcesOptions.concurrentCopies`: The number of concurrent copies used during the synchronization of sources to the remote system. Defaults to `5`. `copySourcesOptions.outputVerbosity`: The verbosity level of source copy operations to the remote system. Accepted levels are `Normal`, `Verbose`, and `Diagnostic`. Defaults to `Normal`.
`rsyncCommandArgs`	A list of command-line arguments passed to `rsync`. Defaults to `[ "-t", "--delete", "--delete-excluded" ]`. This option supports macro expansion and is ignored when targeting WSL1.
`copyBuildOutput`	Specifies whether to copy build output from the remote system back to Windows. Defaults to `false`. This option is ignored when targeting WSL1.

Setting	Description
`copyOptimizations`	An object of options related to source copy optimizations. These options are ignored when targeting WSL1. `copyOptimizations.maxSmallChange`: The maximum number of files to copy using sftp instead of rsync. Defaults to 10. `copyOptimizations.useOptimizations`: Specifies the copy optimizations in use. Accepted values are no copy optimizations (`None`), rsync only optimizations (`RsyncOnly`), or rsync and sftp optimizations (`RsyncAndSftp`). Defaults to `RsyncAndSftp`. `copyOptimizations.rsyncSingleDirectoryCommandArgs`: A list of command-line arguments passed to rsync when copying the contents of a single directory to the remote system. Defaults to `[ "-t", "-d" ]`. This option supports macro expansion.
`copyAdditionalIncludeDirectoriesList`	A list of paths to remote header directories to be copied locally for IntelliSense. This option supports macro expansion.
`copyExcludeDirectoriesList`	A list of paths to remote header directories to not be copied locally for IntelliSense. This option supports macro expansion.
`forceWSL1Toolset`	If `true`, Visual Studio will always use the WSL1 toolset when targeting WSL from Visual Studio. The WSL1 toolset executes all commands locally and relies on Windows drives mounted under the `/mnt` folder to access local source files from WSL. These options may be slower with WSL2. Defaults to `false`. The WSL1 toolset will always be used in Visual Studio 2019 version 16.10. This option will be relevant once native support for WSL2 is available.

Remote pre-build and post-build events

Options for a `remotePrebuildEvent` and `remotePostbuildEvent` have been deprecated with the adoption of *`CMakePresets.json`*.

Encode pre-build, pre-link, and post-build events in your `CMakeLists.txt` using add_custom_command. It ensures the same behavior when building with Visual Studio and from the command line.

If you need behavior that is specific to Visual Studio, you can add a custom remote task in `tasks.vs.json`. To get started, right-click on your root `CMakeLists.txt` in the **Solution Explorer** from **Folder View** and select **Configure Tasks**. You can then add a new remote task to your `tasks.vs.json` file.

The following remote task creates a directory called test on the remote Linux system:

JSON

```
{
      "taskLabel": "mkdir",
      "appliesTo": "CMakeLists.txt",
      "type": "remote",
      "command": "mkdir test",
      "remoteMachineName": "localhost"
  }
```

Right-click on any `CMakeLists.txt` and select the **mkdir** option to execute this task.

The value of `remoteMachineName` must match the Host Name of a connection in the **Connection Manager**.

Microsoft vendor macros

The two Microsoft vendor maps, `Visual Studio Settings` and `Visual Studio Remote Settings`, support all the macros defined by CMake. Our vendor maps support all the macros defined by CMake. For more information, see cmake-presets Macro Expansion. All macros and environment variables are expanded before being passed to CMake.

Visual Studio supports vendor macros with the prefix `ms`. Microsoft vendor macros can only be used in Microsoft vendor maps. CMake can't use presets that have vendor macros outside of a vendor map.

Macro	Description
`$ms{projectDirName}`	Evaluates to the name of the open folder in Visual Studio. This macro is used to set the default value of `sourceDir` in remote copy scenarios. This macro is not supported by Visual Studio Code. Use `${sourceDirName}` instead.

Environment variables

Macro	Description
`$env{<variable-name>}` `$penv{<variable-name>}`	Reference environment variables using this syntax supported by CMake. For more information, see cmake-presets Macro Expansion.

Deprecated macros

A few macros that were supported by *`CMakeSettings.json`* have been deprecated with the adoption of *`CMakePresets.json`*.

Use the macros supported by CMake to construct your file paths. When you use the macros, it ensures that the same *`CMakePresets.json`* file works inside Visual Studio and from the command line.

Deprecated macro	Recommendation
`${projectFile}`	`${sourceDir}/CMakeLists.txt`
`${thisFile}`	`${sourceDir}/CMakePresets.json`

Accepted shell syntax

Use the `$env{HOME}` syntax to reference `$HOME` when constructing Linux paths in the Microsoft vendor maps.

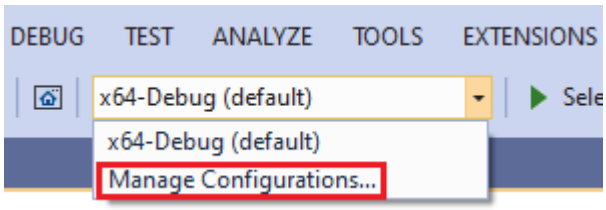
Customize CMake build settings

Article • 12/15/2021

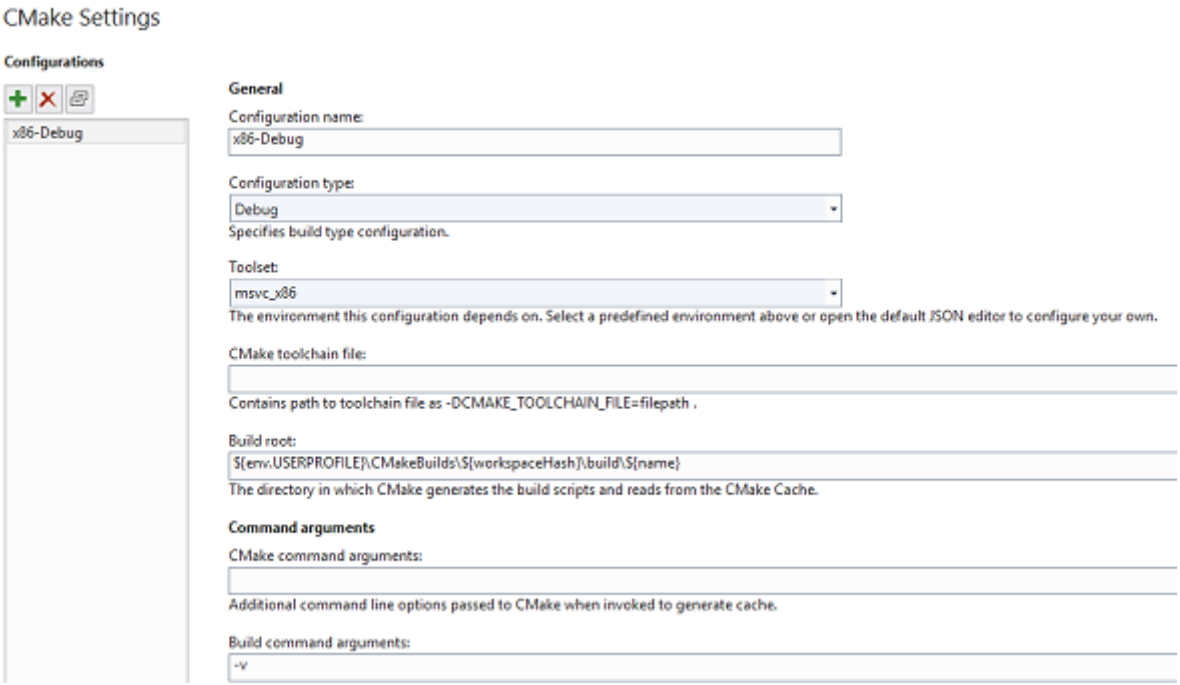
Visual Studio uses a CMake configuration file to drive CMake generation and build. `CMakePresets.json` is supported by Visual Studio 2019 version 16.10 or later and is the recommended CMake configuration file. `CMakePresets.json` is supported directly by CMake and can be used to drive CMake generation and build from Visual Studio, from VS Code, in a Continuous Integration pipeline, and from the command line on Windows, Linux, and Mac. For more information on `CMakePresets.json`, see Configure and build with CMake Presets.

If you maintain projects that use a `CMakeSettings.json` file for CMake build configuration, Visual Studio 2019 and later versions provide a **CMake settings editor**. The editor lets you add CMake configurations and customize their settings easily. It's intended to be a simpler alternative to manually editing the `CMakeSettings.json` file. However, if you prefer to edit the file directly, you can select the **Edit JSON** link in the upper right of the editor.

To open the CMake settings editor, select the **Configuration** drop-down in the main toolbar and choose **Manage Configurations**.



Now you see the **Settings Editor** with the installed configurations on the left.



Visual Studio provides one `x64-Debug` configuration by default. You can add more configurations by choosing the green plus sign. The settings that you see in the editor might vary depending on which configuration is selected.

The options that you choose in the editor are written to a file called `CMakeSettings.json`. This file provides command-line arguments and environment variables that are passed to CMake when you build the projects. Visual Studio never modifies `CMakeLists.txt` automatically; by using `CMakeSettings.json` you can customize the build through Visual Studio while leaving the CMake project files untouched so that others on your team can consume them with whatever tools they're using.

CMake General Settings

The following settings are available under the **General** heading:

Configuration name

Corresponds to the **name** setting. This name appears in the C++ configuration dropdown. You can use the `${name}` macro to compose other property values such as paths.

Configuration type

Corresponds to the **configurationType** setting. Defines the build configuration type for the selected generator. Currently supported values are "Debug", "MinSizeRel", "Release",

and "RelWithDebInfo". It maps to CMAKE_BUILD_TYPE.

Toolset

Corresponds to the **inheritedEnvironments** setting. Defines the compiler environment that's used to build the selected configuration. Supported values depend on the type of configuration. To create a custom environment, choose the **Edit JSON** link in the upper right corner of the Settings editor, and edit the `CMakeSettings.json` file directly.

CMake toolchain file

Path to the CMake toolchain file. This path is passed to CMake as "-DCMAKE_TOOLCHAIN_FILE = <filepath>". Toolchain files specify locations of compilers and toolchain utilities, and other target platform and compiler-related information. By default, Visual Studio uses the vcpkg toolchain file if this setting is unspecified.

Build root

Corresponds to **buildRoot**. Maps to CMAKE_BINARY_DIR, and specifies where to create the CMake cache. The specified folder is created if it doesn't exist.

Command arguments

The following settings are available under the **Command arguments** heading:

CMake command arguments

Corresponds to **cmakeCommandArgs**. Specifies any more command-line options passed to CMake.

Build command arguments

Corresponds to **buildCommandArgs**. Specifies more switches to pass to the underlying build system. For example, passing `-v` when using the Ninja generator forces Ninja to output command lines.

CTest command arguments

Corresponds to **ctestCommandArgs**. Specifies more command-line options to pass to CTest when running tests.

General settings for remote builds

For configurations such as Linux that use remote builds, the following settings are also available:

`rsync` command arguments

Extra command-line options passed to rsync, a fast, versatile file-copying tool.

CMake variables and cache

These settings enable you to set CMake variables and save them in `CMakeSettings.json`. They're passed to CMake at build time, and override whatever values are in the `CMakeLists.txt` file. You can use this section in the same way that you might use the CMakeGUI to view a list of all the CMake variables available to edit. Choose the **Save and generate cache** button to view a list of all CMake variables available to edit, including advanced variables (per the CMakeGUI). You can filter the list by variable name.

Corresponds to **variables**. Contains a name-value pair of CMake variables passed as `-D name=value` to CMake. If your CMake project build instructions specify the addition of any variables directly to the CMake cache file, we recommend you add them here instead.

Advanced settings

CMake generator

Corresponds to **generator**. Maps to the CMake `-G` switch, and specifies the CMake generator to use. This property can also be used as a macro, `${generator}`, when composing other property values. Visual Studio currently supports the following CMake generators:

- "Ninja"
- "Unix Makefiles"
- "Visual Studio 16 2019"

- "Visual Studio 16 2019 Win64"
- "Visual Studio 16 2019 ARM"
- "Visual Studio 15 2017"
- "Visual Studio 15 2017 Win64"
- "Visual Studio 15 2017 ARM"
- "Visual Studio 14 2015"
- "Visual Studio 14 2015 Win64"
- "Visual Studio 14 2015 ARM"

Because Ninja is designed for fast build speeds instead of flexibility and function, it's set as the default. However, some CMake projects may be unable to correctly build using Ninja. If that occurs, you can instruct CMake to generate a Visual Studio project instead.

IntelliSense mode

The IntelliSense mode used by the IntelliSense engine. If no mode is selected, Visual Studio inherits the mode from the specified toolset.

Install directory

The directory in which CMake installs targets. Maps to CMAKE_INSTALL_PREFIX.

CMake executable

The full path to the CMake program executable, including the file name and extension. It allows you to use a custom version of CMake with Visual Studio. For remote builds, specify the CMake location on the remote machine.

For configurations such as Linux that use remote builds, the following settings are also available:

Remote CMakeLists.txt root

The directory on the remote machine that contains the root `CMakeLists.txt` file.

Remote install root

The directory on the remote machine in which CMake installs targets. Maps to CMAKE_INSTALL_PREFIX.

Remote copy sources

Specifies whether to copy source files to the remote machine, and lets you specify whether to use rsync or sftp.

Directly edit CMakeSettings.json

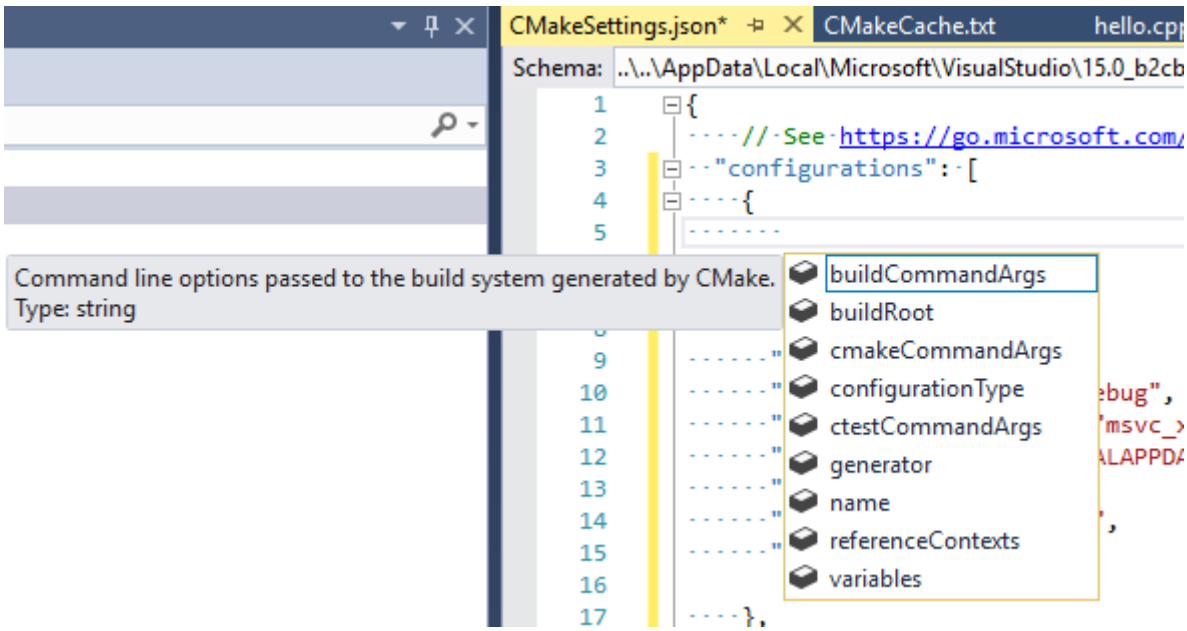
You can also directly edit `CMakeSettings.json` to create custom configurations. The **Settings Editor** has an **Edit JSON** button in the upper right that opens the file for editing.

The following example shows a sample configuration, which you can use as a starting point:

JSON

```
    {
      "name": "x86-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "inheritEnvironments": [ "msvc_x86" ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": ""
    },
```

JSON IntelliSense helps you edit the `CMakeSettings.json` file:



The JSON editor also informs you when you choose incompatible settings.

For more information about each of the properties in the file, see CMakeSettings.json schema reference.

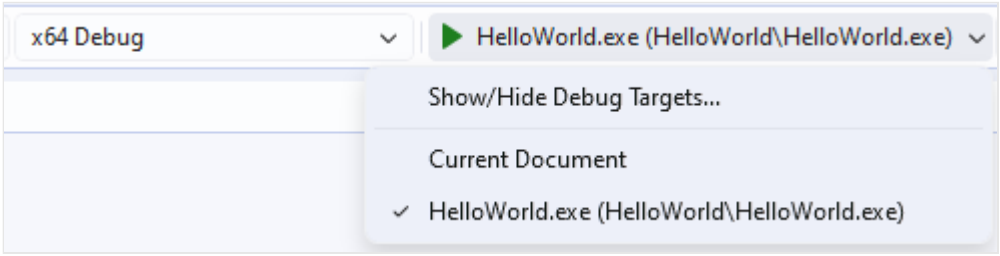
See also

CMake Projects in Visual Studio
Configure a Linux CMake project
Connect to your remote Linux computer
Configure CMake debugging sessions
Deploy, run, and debug your Linux project
CMake predefined configuration reference

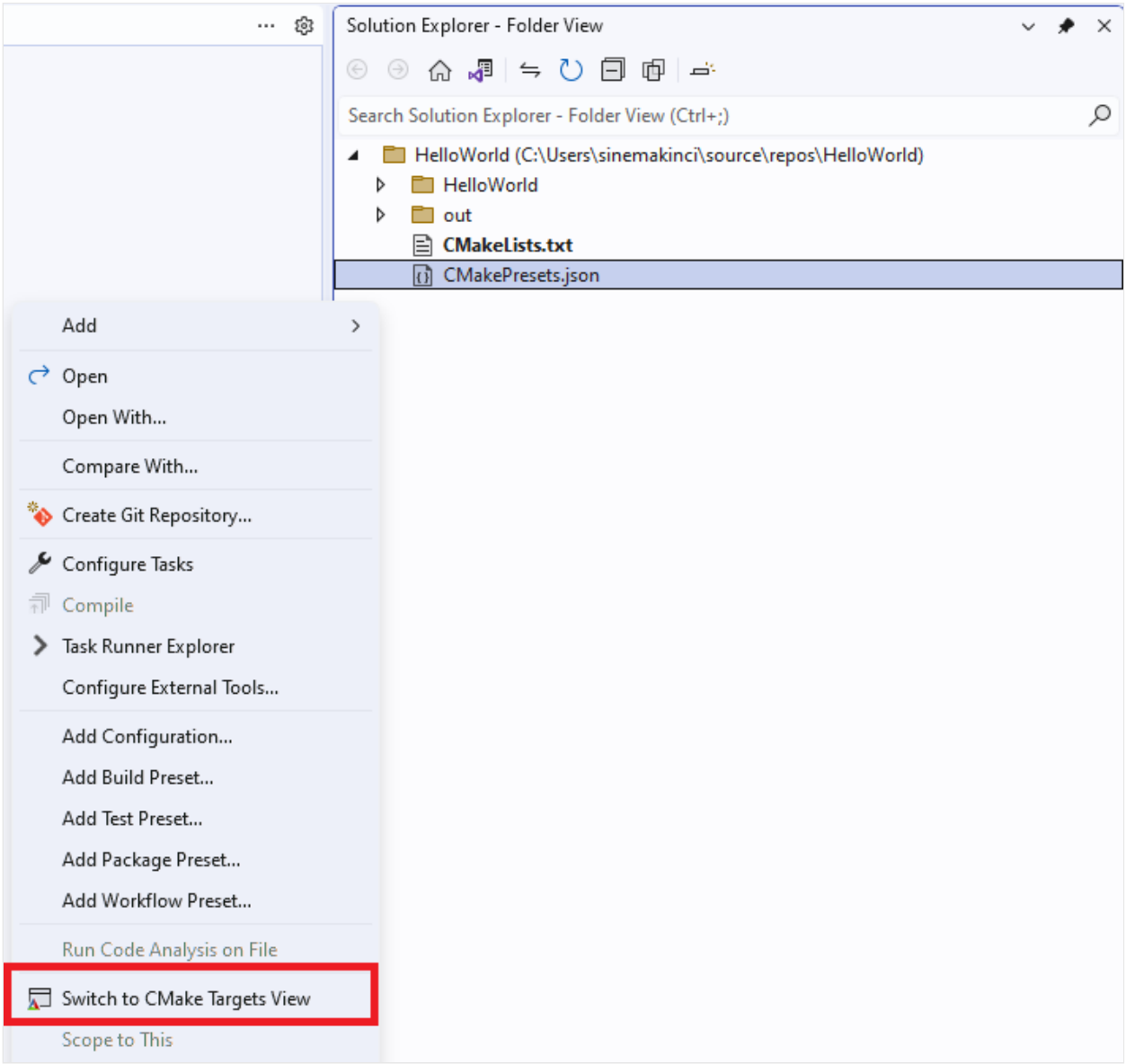
Configure CMake debugging sessions

Article • 10/26/2023

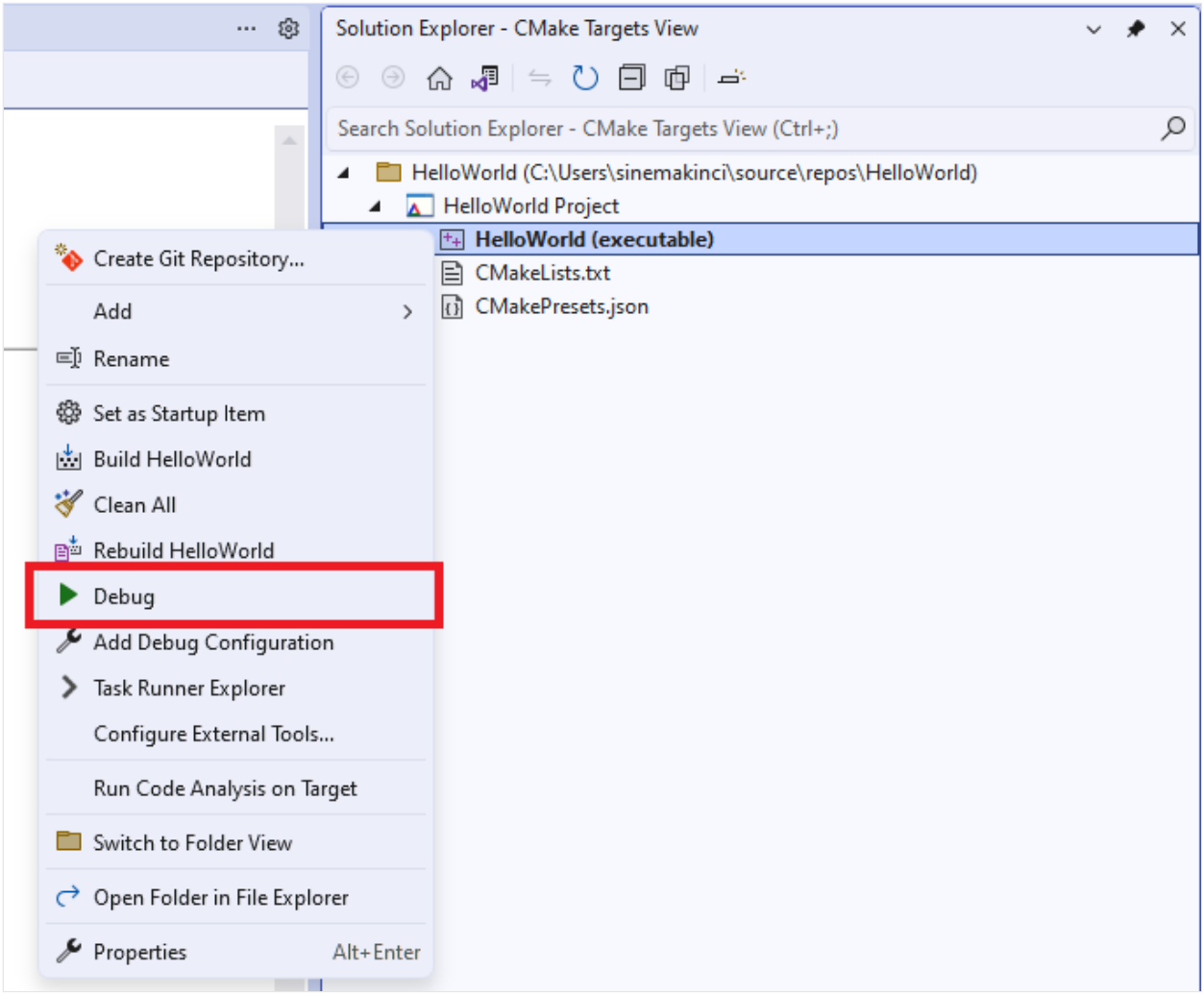
All executable CMake targets are shown in the **Startup Item** dropdown in the toolbar. Select one to start a debugging session and launch the debugger.



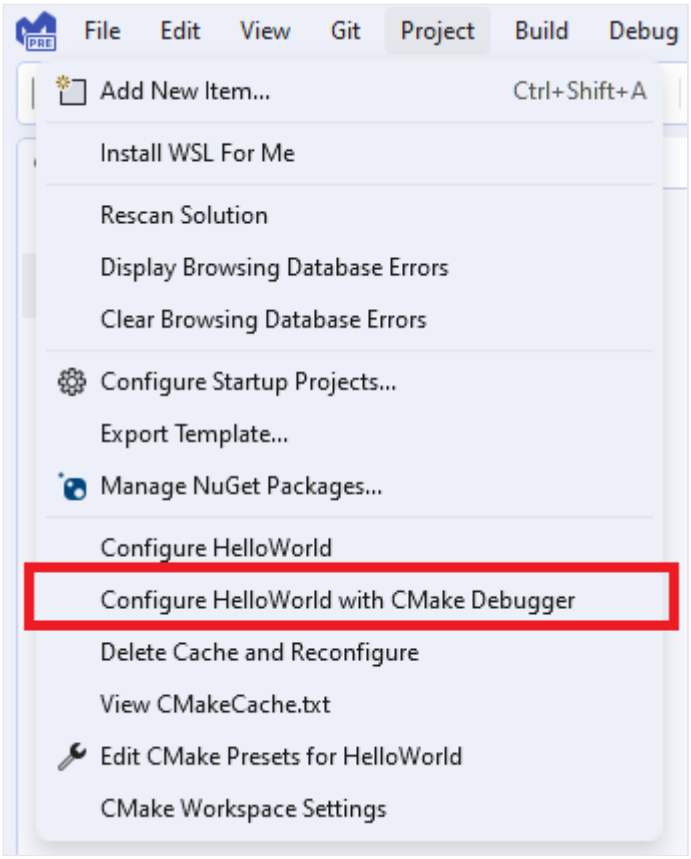
You can also start a debug session from Solution Explorer. First, switch to **CMake Targets View** in the **Solution Explorer** window.



Then, right-click on an executable and select **Debug**. This command automatically starts debugging the selected target based on your active configuration.



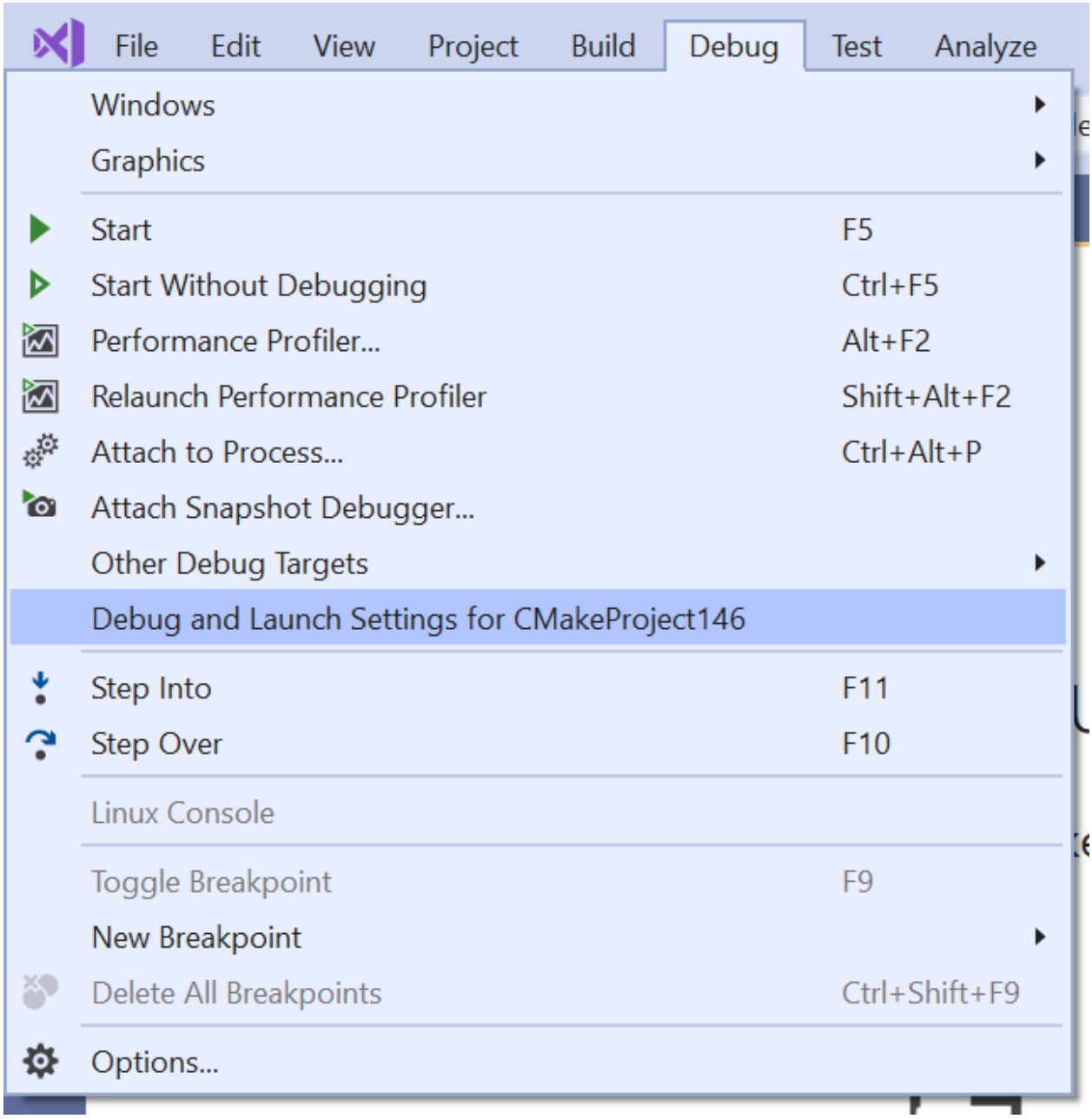
Starting in Visual Studio 2022 Version 17.6, you can also start a debugging session on your CMakeLists.txt file. To do so, just set a breakpoint in your CMakeLists.txt file and run **Configure Project with CMake Debugger** from the **Project** dropdown.



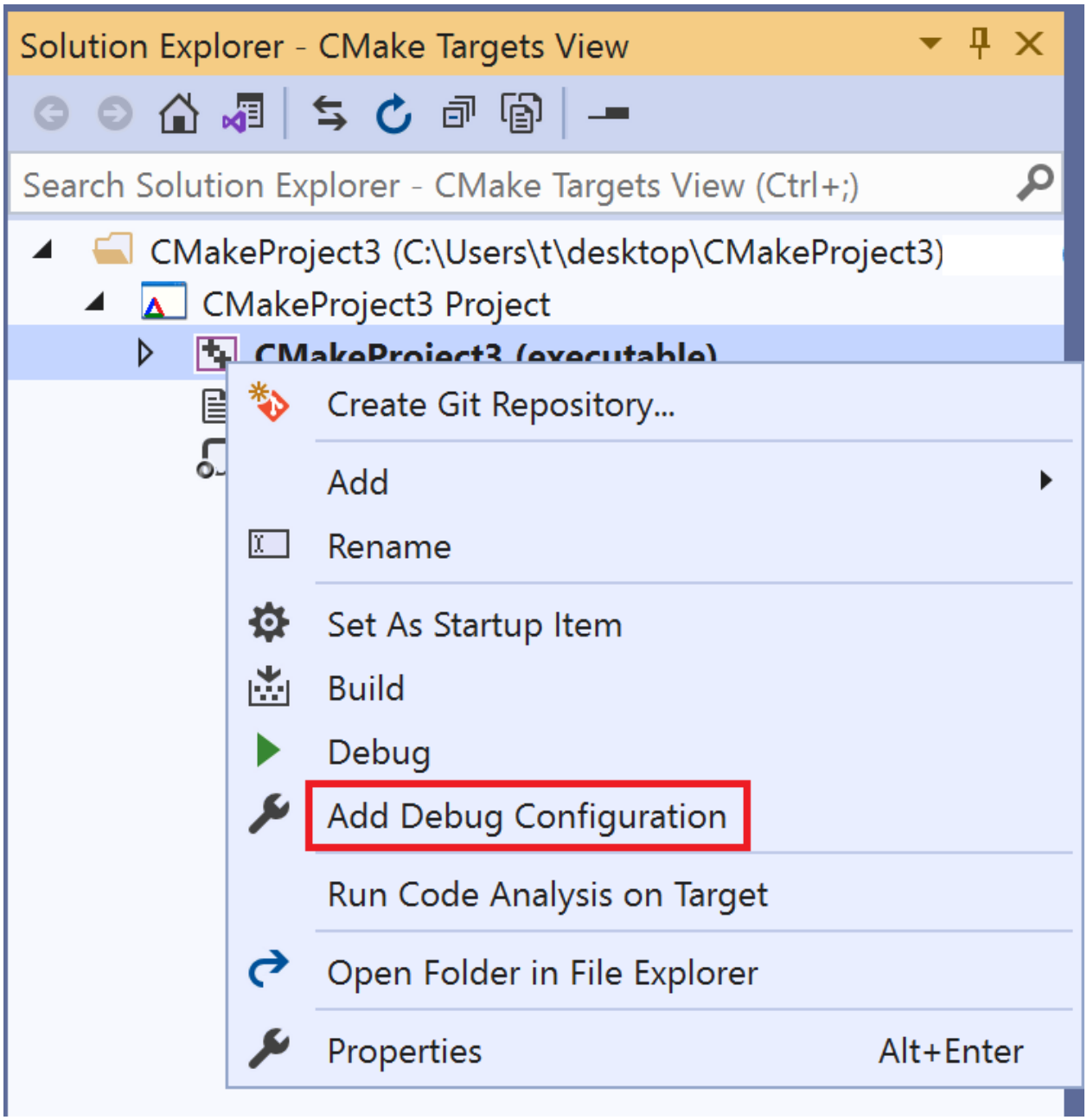
Customize debugger settings

You can customize the debugger settings for any executable CMake target in your project. They're found in a configuration file called *launch.vs.json*, located in a `.vs` folder in your project root. A launch configuration file is useful in most debugging scenarios, because you can configure and save your debugging setup details. There are three entry points to this file:

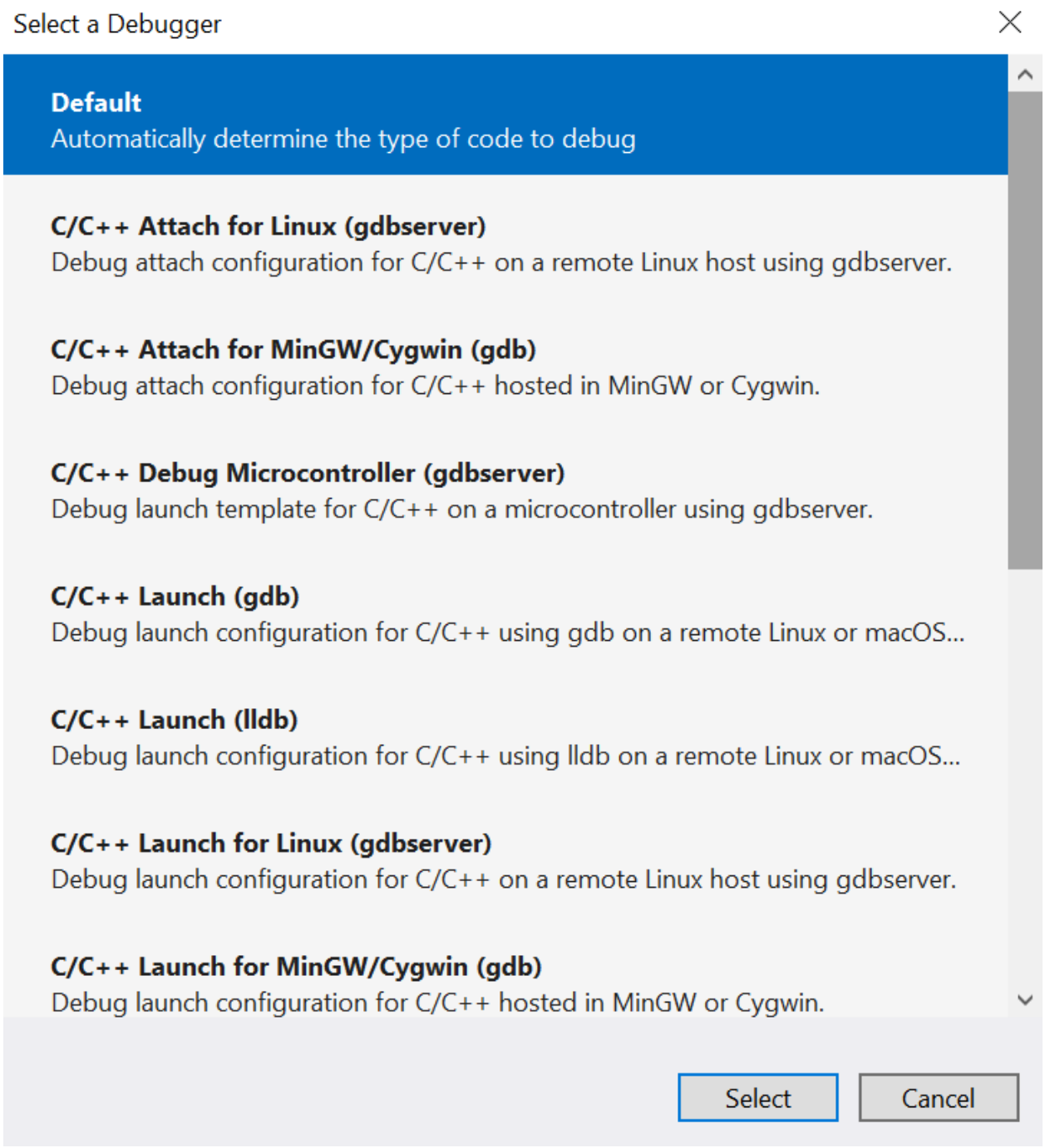
- **Debug Menu:** Select **Debug** > **Debug and Launch Settings for ${activeDebugTarget}** from the main menu to customize the debug configuration specific to your active debug target. If you don't have a debug target selected, this option is grayed out.



- **Targets View:** Navigate to **Targets View** in Solution Explorer. Then, right-click on a debug target and select **Add Debug Configuration** to customize the debug configuration specific to the selected target.



- **Root CMakeLists.txt:** Right-click on a root *CMakeLists.txt* and select **Add Debug Configuration** to open the **Select a Debugger** dialog box. The dialog allows you to add *any* type of debug configuration, but you must manually specify the CMake target to invoke via the `projectTarget` property.



You can edit the *launch.vs.json* file to create debug configurations for any number of CMake targets. When you save the file, Visual Studio creates an entry for each new configuration in the **Startup Item** dropdown.

Reference keys in CMakeSettings.json

To reference any key in a *CMakeSettings.json* file, prepend `cmake.` to it in *launch.vs.json*. The following example shows a simple *launch.vs.json* file that pulls in the value of the `remoteCopySources` key in the *CMakeSettings.json* file for the currently selected configuration:

JSON

```
{
  "version": "0.2.1",
  "configurations": [
    {
      "type": "default",
      "project": "CMakeLists.txt",
      "projectTarget": "CMakeHelloWorld.exe (Debug\\CMakeHelloWorld.exe)",
      "name": "CMakeHelloWorld.exe (Debug\\CMakeHelloWorld.exe)",
      "args": ["${cmake.remoteCopySources}"]
    }
  ]
}
```

Environment variables defined in *CMakeSettings.json* can also be used in launch.vs.json using the syntax `${env.VARIABLE_NAME}`. In Visual Studio 2019 version 16.4 and later, debug targets are automatically launched using the environment you specify in *CMakeSettings.json*. You can unset an environment variable by setting it to **null**.

Launch.vs.json reference

There are many *launch.vs.json* properties to support all your debugging scenarios. The following properties are common to all debug configurations, both remote and local:

- `projectTarget`: Specifies the CMake target to invoke when building the project. Visual Studio autopopulates this property if you enter *launch.vs.json* from the **Debug Menu** or **Targets View**. This value must match the name of an existing debug target listed in the **Startup Item** dropdown.
- `env`: Additional environment variables to add using the syntax:

 JSON

  ```
  "env": {
        "DEBUG_LOGGING_LEVEL": "trace;info",
        "ENABLE_TRACING": "true"
     }
  ```

- `args`: Command-line arguments passed to the program to debug.

Launch.vs.json reference for remote projects and WSL

In Visual Studio 2019 version 16.6, we added a new debug configuration of `type: cppgdb` to simplify debugging on remote systems and WSL. Old debug configurations of `type: cppdbg` are still supported.

Configuration type `cppgdb`

- `name`: A friendly name to identify the configuration in the **Startup Item** dropdown.
- `project`: Specifies the relative path to the project file. Normally, you don't need to change this path when debugging a CMake project.
- `projectTarget`: Specifies the CMake target to invoke when building the project. Visual Studio autopopulates this property if you enter *launch.vs.json* from the **Debug Menu** or **Targets View**. This target value must match the name of an existing debug target listed in the **Startup Item** dropdown.
- `debuggerConfiguration`: Indicates which set of debugging default values to use. In Visual Studio 2019 version 16.6, the only valid option is `gdb`. Visual Studio 2019 version 16.7 or later also supports `gdbserver`.
- `args`: Command-line arguments passed on startup to the program being debugged.
- `env`: Additional environment variables passed to the program being debugged. For example, `{"DISPLAY": "0.0"}`.
- `processID`: Linux process ID to attach to. Only used when attaching to a remote process. For more information, see Troubleshoot attaching to processes using GDB.

Additional options for the `gdb` configuration

- `program`: Defaults to `"${debugInfo.fullTargetPath}"`. The Unix path to the application to debug. Only required if different than the target executable in the build or deploy location.
- `remoteMachineName`: Defaults to `"${debugInfo.remoteMachineName}"`. Name of the remote system that hosts the program to debug. Only required if different than the build system. Must have an existing entry in the Connection Manager. Press **Ctrl+Space** to view a list of all existing remote connections.
- `cwd`: Defaults to `"${debugInfo.defaultWorkingDirectory}"`. The Unix path to the directory on the remote system where `program` is run. The directory must exist.
- `gdbpath`: Defaults to `/usr/bin/gdb`. Full Unix path to the `gdb` used to debug. Only required if using a custom version of `gdb`.
- `preDebugCommand`: A Linux command to run immediately before invoking `gdb`. `gdb` doesn't start until the command completes. You can use the option to run a script

before the execution of `gdb`.

Additional options allowed with the `gdbserver` configuration (16.7 or later)

- `program`: Defaults to `"${debugInfo.fullTargetPath}"`. The Unix path to the application to debug. Only required if different than the target executable in the build or deploy location.

 > **Tip**
 >
 > Deploy is not yet supported for local cross-compilation scenarios. If you are cross-compiling on Windows (for example, using a cross-compiler on Windows to build a Linux ARM executable) then you'll need to manually copy the binary to the location specified by `program` on the remote ARM machine before debugging.

- `remoteMachineName`: Defaults to `"${debugInfo.remoteMachineName}"`. Name of the remote system that hosts the program to debug. Only required if different than the build system. Must have an existing entry in the Connection Manager. Press **Ctrl+Space** to view a list of all existing remote connections.

- `cwd`: Defaults to `"${debugInfo.defaultWorkingDirectory}"`. Full Unix path to the directory on the remote system where `program` is run. The directory must exist.

- `gdbPath`: Defaults to `${debugInfo.vsInstalledGdb}`. Full Windows path to the `gdb` used to debug. Defaults to the `gdb` installed with the Linux development with C/C++ workload.

- `gdbserverPath`: Defaults to `usr/bin/gdbserver`. Full Unix path to the `gdbserver` used to debug.

- `preDebugCommand`: A Linux command to run immediately before starting `gdbserver`. `gdbserver` doesn't start until the command completes.

Deployment options

Use the following options to separate your build machine (defined in CMakeSettings.json) from your remote debug machine.

- `remoteMachineName`: Remote debug machine. Only required if different than the build machine. Must have an existing entry in the Connection Manager. Press **Ctrl+Space** to view a list of all existing remote connections.
- `disableDeploy`: Defaults to `false`. Indicates whether build/debug separation is disabled. When `false`, this option allows build and debug to occur on two separate machines.
- `deployDirectory`: Full Unix path to the directory on `remoteMachineName` that the executable gets copied to.
- `deploy`: An array of advanced deployment settings. You only need to configure these settings when you want more granular control over the deployment process. By default, only the files necessary for the process to debug get deployed to the remote debug machine.
 - `sourceMachine`: The machine from which the file or directory is copied. Press **Ctrl+Space** to view a list of all the remote connections stored in the Connection Manager. When building natively on WSL, this option is ignored.
 - `targetMachine`: The machine to which the file or directory is copied. Press **Ctrl+Space** to view a list of all the remote connections stored in the Connection Manager.
 - `sourcePath`: The file or directory location on `sourceMachine`.
 - `targetPath`: The file or directory location on `targetMachine`.
 - `deploymentType`: A description of the deployment type. `LocalRemote` and `RemoteRemote` are supported. `LocalRemote` means copying from the local file system to the remote system specified by `remoteMachineName` in *launch.vs.json*. `RemoteRemote` means copying from the remote build system specified in *CMakeSettings.json* to the different remote system specified in *launch.vs.json*.
 - `executable`: Indicates whether the deployed file is an executable.

Execute custom `gdb` commands

Visual Studio supports executing custom `gdb` commands to interact with the underlying debugger directly. For more information, see Executing custom gdb lldb commands.

Enable logging

Enable MIEngine logging to see what commands get sent to `gdb`, what output `gdb` returns, and how long each command takes. Learn more

Configuration type `cppdbg`

The following options can be used when debugging on a remote system or WSL using the `cppdbg` configuration type. In Visual Studio 2019 version 16.6 or later, configuration type `cppgdb` is recommended.

- `name`: A friendly name to identify the configuration in the **Startup Item** dropdown.
- `project`: Specifies the relative path to the project file. Normally, you don't need to change this value when debugging a CMake project.
- `projectTarget`: Specifies the CMake target to invoke when building the project. Visual Studio autopopulates this property if you enter *launch.vs.json* from the **Debug Menu** or **Targets View**. This value must match the name of an existing debug target listed in the **Startup Item** dropdown.
- `args`: Command-line arguments passed on startup to the program being debugged.
- `processID`: Linux process ID to attach to. Only used when attaching to a remote process. For more information, see Troubleshoot attaching to processes using GDB.
- `program`: Defaults to `"${debugInfo.fullTargetPath}"`. The Unix path to the application to debug. Only required if different than the target executable in the build or deploy location.
- `remoteMachineName`: Defaults to `"${debugInfo.remoteMachineName}"`. Name of the remote system that hosts the program to debug. Only required if different than the build system. Must have an existing entry in the Connection Manager. Press **Ctrl+Space** to view a list of all existing remote connections.
- `cwd`: Defaults to `"${debugInfo.defaultWorkingDirectory}"`. Full Unix path to the directory on the remote system where `program` is run. The directory must exist.
- `environment`: Additional environment variables passed to the program being debugged. For example,

JSON

```
  "environment": [
      {
        "name": "ENV1",
        "value": "envvalue1"
      },
      {
        "name": "ENV2",
```

```
        "value": "envvalue2"
      }
    ]
```

- `pipeArgs`: An array of command-line arguments passed to the pipe program to configure the connection. The pipe program is used to relay standard input/output between Visual Studio and `gdb`. Most of this array **doesn't need to be customized** when debugging CMake projects. The exception is the `${debuggerCommand}`, which launches `gdb` on the remote system. It can be modified to:

 - Export the value of the environment variable DISPLAY on your Linux system. In the following example, this value is `:1`.

 JSON

    ```
    "pipeArgs": [
        "/s",
        "${debugInfo.remoteMachineId}",
        "/p",
        "${debugInfo.parentProcessId}",
        "/c",
        "export DISPLAY=:1;${debuggerCommand}",
        "--tty=${debugInfo.tty}"
      ],
    ```

 - Run a script before the execution of `gdb`. Ensure execute permissions are set on your script.

 JSON

    ```
    "pipeArgs": [
        "/s",
        "${debugInfo.remoteMachineId}",
        "/p",
        "${debugInfo.parentProcessId}",
        "/c",
        "/path/to/script.sh;${debuggerCommand}",
        "--tty=${debugInfo.tty}"
      ],
    ```

- `stopOnEntry`: A boolean that specifies whether to break as soon as the process is launched. The default is false.

- `visualizerFile`: A .natvis file to use when debugging this process. This option is incompatible with `gdb` pretty printing. Also set `showDisplayString` when you set this property.

- `showDisplayString`: A boolean that enables the display string when a `visualizerFile` is specified. Setting this option to `true` can cause slower performance during debugging.
- `setupCommands`: One or more `gdb` command(s) to execute, to set up the underlying debugger.
- `miDebuggerPath`: The full path to `gdb`. When unspecified, Visual Studio searches PATH first for the debugger.
- Finally, all of the deployment options defined for the `cppgdb` configuration type can be used by the `cppdbg` configuration type as well.

Debug using `gdbserver`

You can configure the `cppdbg` configuration to debug using `gdbserver`. You can find more details and a sample launch configuration in the Microsoft C++ Team Blog post Debugging Linux CMake Projects with gdbserver.

See also

CMake projects in Visual Studio
Configure a Linux CMake project
Connect to your remote Linux computer
Customize CMake build settings
Configure CMake debugging sessions
Deploy, run, and debug your Linux project
CMake predefined configuration reference

`CMakeSettings.json` schema reference

Article • 02/24/2023

The *`CMakeSettings.json`* file contains information that Visual Studio uses for IntelliSense and to construct the command-line arguments that it passes to CMake for a specified configuration and compiler environment. A *configuration* specifies properties that apply to a specific platform and build-type, for example, `x86-Debug` or `Linux-Release`. Each configuration specifies an *environment*, which encapsulates information about the compiler toolset, for example MSVC, GCC, or Clang. CMake uses the command-line arguments to regenerate the root *`CMakeCache.txt`* file and other project files for the project. The values can be overridden in the *`CMakeLists.txt`* files.

You can add or remove configurations in the IDE and then edit them directly in the JSON file or use the **CMake Settings editor** (Visual Studio 2019 and later). You can switch between the configurations easily in the IDE to generate the various project files. For more information, see Customize CMake build settings in Visual Studio.

Configurations

The `configurations` array contains all the configurations for a CMake project. For more information about the pre-defined configurations, see CMake predefined configuration reference. You can add any number of pre-defined or custom configurations to the file.

A `configuration` has these properties:

- `addressSanitizerEnabled`: If **`true`**, compiles the program using AddressSanitizer. On Linux, compile with `-fno-omit-frame-pointer` and compiler optimization level `-Os` or `-Oo` for best results.
- `addressSanitizerRuntimeFlags`: The runtime flags passed to AddressSanitizer in the `ASAN_OPTIONS` environment variable. Format: flag1=value:flag2=value2.
- `buildCommandArgs`: Specifies native build switches passed to CMake after `--build --`. For example, passing `-v` when using the Ninja generator forces Ninja to output command lines. For more information on Ninja commands, see Ninja command line arguments.
- `buildRoot`: Specifies the directory in which CMake generates build scripts for the chosen generator. Maps to **`-DCMAKE_BINARY_DIR`** switch and specifies where *`CMakeCache.txt`* is created. If the folder doesn't exist, it's created. Supported

macros include `${workspaceRoot}`, `${workspaceHash}`, `${projectFile}`, `${projectDir}`, `${thisFile}`, `${thisFileDir}`, `${name}`, `${generator}`, `${env.VARIABLE}`.

- `cacheGenerationCommand`: Specifies a command-line tool and arguments, for example **`gencache.bat debug`** to generate the cache. The command is run from the shell in the specified environment for the configuration when the user explicitly requests regeneration, or a *`CMakeLists.txt`* or *`CMakeSettings.json`* file is modified.
- `cacheRoot`: Specifies the path to a CMake cache. This directory should contain an existing *`CMakeCache.txt`* file.
- `clangTidyChecks`: comma-separated list of warnings that's passed to clang-tidy; wildcards are allowed and a '-' prefix removes checks.
- `cmakeCommandArgs`: Specifies any extra command-line options to pass to CMake when invoked to generate the project files.
- `cmakeToolchain`: Specifies the toolchain file. It's passed to CMake using `-DCMAKE_TOOLCHAIN_FILE`.
- `codeAnalysisRuleset`: Specifies the ruleset to use when running code analysis. You can use a full path or the filename of a ruleset file installed by Visual Studio.
- `configurationType`: Specifies the build type configuration for the selected generator. May be one of:
 - `Debug`
 - `Release`
 - `MinSizeRel`
 - `RelWithDebInfo`
- `ctestCommandArgs`: Specifies any extra command-line options to pass to CTest when running the tests.
- `description`: The description of this configuration that appears in menus.
- `enableClangTidyCodeAnalysis`: Use Clang-Tidy for code analysis.
- `enableMicrosoftCodeAnalysis`: Use Microsoft code analysis tools for code analysis.
- `generator`: Specifies the CMake generator to use for this configuration. May be one of:

 Visual Studio 2019 only:

- `Visual Studio 16 2019`
- `Visual Studio 16 2019 Win64`
- `Visual Studio 16 2019 ARM`

Visual Studio 2017 and later:

- `Visual Studio 15 2017`
- `Visual Studio 15 2017 Win64`
- `Visual Studio 15 2017 ARM`
- `Visual Studio 14 2015`
- `Visual Studio 14 2015 Win64`
- `Visual Studio 14 2015 ARM`
- `Unix Makefiles`
- `Ninja`

Because Ninja is designed for fast build speeds instead of flexibility and function, it's set as the default. However, some CMake projects may be unable to correctly build using Ninja. If a build failure occurs, you can instruct CMake to generate Visual Studio projects instead.

To specify a Visual Studio generator in Visual Studio 2017, open the settings editor from the main menu by choosing **CMake | Change CMake Settings**. Delete "Ninja" and enter "V". This change activates IntelliSense, which lets you choose the generator you want.

To specify a Visual Studio generator in Visual Studio 2019, right-click on the *`CMakeLists.txt`* file in **Solution Explorer** and choose **CMake Settings for project** > **Show Advanced Settings** > **CMake Generator**.

By default, when the active configuration specifies a Visual Studio generator, it invokes MSBuild with `-m -v:minimal` arguments. To customize the build, use the `buildCommandArgs` property inside the *`CMakeSettings.json`* file. Here, you can specify MSBuild command line arguments to pass to the build system:

JSON

```
"buildCommandArgs": "-m:8 -v:minimal -p:PreferredToolArchitecture=x64"
```

- `installRoot`: Specifies the directory in which CMake generates install targets for the chosen generator. Supported macros include `${workspaceRoot}`, `${workspaceHash}`, `${projectFile}`, `${projectDir}`, `${thisFile}`, `${thisFileDir}`, `${name}`, `${generator}`, `${env.VARIABLE}`.

- `inheritEnvironments`: Specifies one or more compiler environments that this configuration depends on. May be any custom environment or one of the predefined environments. For more information, see Environments.
- `intelliSenseMode`: Specifies the mode used for computing intellisense information". The value may be one of:
 - `windows-msvc-x86`
 - `windows-msvc-x64`
 - `windows-msvc-arm`
 - `windows-msvc-arm64`
 - `android-clang-x86`
 - `android-clang-x64`
 - `android-clang-arm`
 - `android-clang-arm64`
 - `ios-clang-x86`
 - `ios-clang-x64`
 - `ios-clang-arm`
 - `ios-clang-arm64`
 - `windows-clang-x86`
 - `windows-clang-x64`
 - `windows-clang-arm`
 - `windows-clang-arm64`
 - `linux-gcc-x86`
 - `linux-gcc-x64`
 - `linux-gcc-arm`
- `name`: names the configuration. For more information about the pre-defined configurations, see CMake predefined configuration reference.
- `wslPath`: the path to the launcher of an instance of Windows Subsystem for Linux.

Settings for CMake Linux projects

- `remoteMachineName`: Specifies the name of the remote Linux machine that hosts CMake, builds, and the debugger. Use the Connection Manager for adding new Linux machines. Supported macros include `${defaultRemoteMachineName}`.
- `remoteCopySourcesOutputVerbosity`: Specifies the verbosity level of the source copying operation to the remote machine. May be one of `Normal`, `Verbose`, or `Diagnostic`.

- `remoteCopySourcesConcurrentCopies`: Specifies the concurrent copies to use during synchronization of the sources to the remote machine (sftp only).
- `remoteCopySourcesMethod`: Specifies the method to copy files to the remote machine. May be `rsync` or `sftp`.
- `remoteCMakeListsRoot`: Specifies the directory on the remote machine that contains the CMake project. Supported macros include `${workspaceRoot}`, `${workspaceHash}`, `${projectFile}`, `${projectDir}`, `${thisFile}`, `${thisFileDir}`, `${name}`, `${generator}`, and `${env.VARIABLE}`.
- `remoteBuildRoot`: Specifies the directory on the remote machine in which CMake generates build scripts for the chosen generator. Supported macros include `${workspaceRoot}`, `${workspaceHash}`, `${projectFile}`, `${projectDir}`, `${thisFile}`, `${thisFileDir}`, `${name}`, `${generator}`, `${env.VARIABLE}`.
- `remoteInstallRoot`: Specifies the directory on the remote machine in which CMake generates install targets for the chosen generator. Supported macros include `${workspaceRoot}`, `${workspaceHash}`, `${projectFile}`, `${projectDir}`, `${thisFile}`, `${thisFileDir}`, `${name}`, `${generator}`, and `${env.VARIABLE}`, where `VARIABLE` is an environment variable that's been defined at the system, user, or session level.
- `remoteCopySources`: A `boolean` that specifies whether Visual Studio should copy source files to the remote machine. The default is true. Set to false if you manage file synchronization yourself.
- `remoteCopyBuildOutput`: A `boolean` that specifies whether to copy the build outputs from the remote system.
- `remoteCopyAdditionalIncludeDirectories`: Additional include directories to be copied from the remote machine to support IntelliSense. Format as "/path1;/path2...".
- `remoteCopyExcludeDirectories`: Include directories NOT to copy from the remote machine. Format as "/path1;/path2...".
- `remoteCopyUseCompilerDefaults`: Specifies whether to use the compiler's default defines and include paths for IntelliSense. Should only be false if the compilers in use to not support gcc-style arguments.
- `rsyncCommandArgs`: Specifies a set of command-line options passed to rsync.
- `remoteCopySourcesExclusionList`: An `array` that specifies a list of paths to be excluded when copying source files: a path can be the name of a file/directory, or a relative path from the root of the copy. Wildcards `*` and `?` can be used for glob pattern matching.
- `cmakeExecutable`: Specifies the full path to the CMake program executable, including the file name and extension.
- `remotePreGenerateCommand`: Specifies the command to run before running CMake to parse the *`CMakeLists.txt`* file.

- `remotePrebuildCommand`: Specifies the command to run on the remote machine before building.
- `remotePostbuildCommand`: Specifies the command to run on the remote machine after building.
- `variables`: Contains a name-value pair of CMake variables that get passed as `-D name=value` to CMake. If your CMake project build instructions specify the addition of any variables directly to the *`CMakeCache.txt`* file, we recommend you add them here instead. This example shows how to specify the name-value pairs to use the 14.14.26428 MSVC toolset:

JSON

```
"variables": [
    {
      "name": "CMAKE_CXX_COMPILER",
      "value": "C:/Program Files (x86)/Microsoft Visual
Studio/157/Enterprise/VC/Tools/MSVC/14.14.26428/bin/HostX86/x86/cl.exe",
      "type": "FILEPATH"
    },
    {
      "name": "CMAKE_C_COMPILER",
      "value": "C:/Program Files (x86)/Microsoft Visual
Studio/157/Enterprise/VC/Tools/MSVC/14.14.26428/bin/HostX86/x86/cl.exe",
      "type": "FILEPATH"
    }
  ]
```

If you don't define the `"type"`, the `"STRING"` type is assumed by default.

- `remoteCopyOptimizations`: **Visual Studio 2019 version 16.5 or later** properties for controlling source copy to the remote target. Optimizations are enabled by default. Includes `remoteCopyUseOptimizations`, `rsyncSingleDirectoryCommandArgs`, and `remoteCopySourcesMaxSmallChange`.

Environments

An *environment* encapsulates the environment variables set in the process that Visual Studio uses to invoke CMake. For MSVC projects, it captures the variables set in a developer command prompt for a specific platform. For example, the `msvc_x64_x64` environment is the same as running the **Developer Command Prompt for VS {version}** with the **-arch=amd64 -host_arch=amd64** arguments. You can use the `env.{<variable_name>}` syntax in *`CMakeSettings.json`* to reference the individual environment

variables, for example to construct paths to folders. The following predefined environments are provided:

- `linux_arm`: Target ARM Linux remotely.
- `linux_x64`: Target x64 Linux remotely.
- `linux_x86`: Target x86 Linux remotely.
- `msvc_arm`: Target ARM Windows with the MSVC compiler.
- `msvc_arm_x64`: Target ARM Windows with the 64-bit MSVC compiler.
- `msvc_arm64`: Target ARM64 Windows with the MSVC compiler.
- `msvc_arm64_x64`: Target ARM64 Windows with the 64-bit MSVC compiler.
- `msvc_arm64ec`: Target ARM64EC Windows with the MSVC compiler.
- `msvc_arm64ec_x64`: Target ARM64EC Windows with the 64-bit MSVC compiler.
- `msvc_x64`: Target x64 Windows with the MSVC compiler.
- `msvc_x64_x64`: Target x64 Windows with the 64-bit MSVC compiler.
- `msvc_x86`: Target x86 Windows with the MSVC compiler.
- `msvc_x86_x64`: Target x86 Windows with the 64-bit MSVC compiler.

Accessing environment variables from *`CMakeLists.txt`*

From a *`CMakeLists.txt`* file, all environment variables are referenced by the syntax `$ENV{variable_name}`. To see the available variables for an environment, open the corresponding command prompt and type `SET`. Some of the information in environment variables is also available through CMake system introspection variables, but you may find it more convenient to use the environment variable. For example, you can easily retrieve the MSVC compiler version or Windows SDK version through the environment variables.

Custom environment variables

In *`CMakeSettings.json`*, you can define custom environment variables globally or per-configuration in the `environments` array. A custom environment is a convenient way to group a set of properties. You can use it in place of a predefined environment, or to extend or modify a predefined environment. Each item in the `environments` array consists of:

- `namespace`: Names the environment so that its variables can be referenced from a configuration in the form `namespace.variable`. The default environment object is called `env` and is populated with certain system environment variables including `%USERPROFILE%`.

- `environment`: Uniquely identifies this group of variables. Allows the group to be inherited later in an `inheritEnvironments` entry.
- `groupPriority`: An integer that specifies the priority of these variables when evaluating them. Higher number items are evaluated first.
- `inheritEnvironments`: An array of values that specify the set of environments that are inherited by this group. This feature lets you inherit default environments and create custom environment variables to pass to CMake when it runs.

Visual Studio 2019 version 16.4 and later: Debug targets are automatically launched with the environment you specify in *`CMakeSettings.json`*. You can override or add environment variables on a per-target or per-task basis in launch.vs.json and tasks.vs.json.

The following example defines one global variable, `BuildDir`, which is inherited in both the x86-Debug and x64-Debug configurations. Each configuration uses the variable to specify the value for the `buildRoot` property for that configuration. Note also how each configuration uses the `inheritEnvironments` property to specify a variable that applies only to that configuration.

JSON

```
{
  // The "environments" property is an array of key-value pairs of the form
  // { "EnvVar1": "Value1", "EnvVar2": "Value2" }
  "environments": [
    {
      "BuildDir":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build",
    }
  ],

  "configurations": [
    {
      "name": "x86-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      // Inherit the defaults for using the MSVC x86 compiler.
      "inheritEnvironments": [ "msvc_x86" ],
      "buildRoot": "${env.BuildDir}\\${name}"    },
    {
      "name": "x64-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      // Inherit the defaults for using the MSVC x64 compiler.
      "inheritEnvironments": [ "msvc_x64" ],
      "buildRoot": "${env.BuildDir}\\${name}"
    }
```

```
  ]
}
```

In the next example, the x86-Debug configuration defines its own value for the **BuildDir** property. This value overrides the value set by the global **BuildDir** property so that **BuildRoot** evaluates to `D:\custom-builddir\x86-Debug`.

JSON

```
{
  "environments": [
    {
      "BuildDir": "${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}",
    }
  ],

  "configurations": [
    {
      "name": "x86-Debug",

      // The syntax for this property is the same as the global one above.
      "environments": [
        {
          // Replace the global property entirely.
          "BuildDir": "D:\\custom-builddir"
          // This environment does not specify a namespace, hence by default
"env" is assumed.
          // "namespace" : "name" would require that this variable be
referenced with "${name.BuildDir}".
        }
      ],

      "generator": "Ninja",
      "configurationType": "Debug",
      "inheritEnvironments": [ "msvc_x86" ],
      // Evaluates to "D:\custom-builddir\x86-Debug"
      "buildRoot": "${env.BuildDir}\\${name}"
    },
    {
      "name": "x64-Debug",

      "generator": "Ninja",
      "configurationType": "Debug",
      "inheritEnvironments": [ "msvc_x64" ],
      // Since this configuration doesn't modify BuildDir, it inherits
      // from the one defined globally.
      "buildRoot": "${env.BuildDir}\\${name}"
    }
  ]
}
```

Macros

The following macros can be used in `CMakeSettings.json`:

- `${workspaceRoot}` – the full path of the workspace folder
- `${workspaceHash}` – hash of workspace location; useful for creating a unique identifier for the current workspace (for example, to use in folder paths)
- `${projectFile}` – the full path of the root `CMakeLists.txt` file
- `${projectDir}` – the full path of the folder containing the root `CMakeLists.txt` file
- `${projectDirName}` – the name of the folder containing the root `CMakeLists.txt` file
- `${thisFile}` – the full path of the `CMakeSettings.json` file
- `${name}` – the name of the configuration
- `${generator}` – the name of the CMake generator used in this configuration

All references to macros and environment variables in `CMakeSettings.json` are expanded before being passed to the CMake command line.

Ninja command-line arguments

If targets are unspecified, Ninja builds the 'default' target.

Windows Command Prompt

```
C:\Program Files (x86)\Microsoft Visual Studio\Preview\Enterprise>ninja -?
ninja: invalid option -- `-?'
usage: ninja [options] [targets...]
```

Option	Description
`--version`	Print ninja version ("1.7.1")
`-C DIR`	Change to DIR before doing anything else
`-f FILE`	Specify input build file (default=`build.ninja`)
`-j N`	Run `N` jobs in parallel (default=14, derived from CPUs available)
`-k N`	Keep going until `N` jobs fail (default=1)
`-l N`	Don't start new jobs if the load average is greater than `N`

Option	Description
`-n`	Dry run (don't run commands but act like they succeeded)
`-v`	Show all command lines while building
`-d MODE`	Enable debugging (use `-d list` to list modes)
`-t TOOL`	Run a subtool (use `-t list` to list subtools). Ends any top-level options; further flags are passed to the tool
`-w FLAG`	Adjust warnings (use `-w list` to list warnings)

CMake predefined build configurations

Article • 10/29/2021

In a CMake project, build configurations are stored in a `CMakeSettings.json` file. When you choose **Manage Configurations** from the build configuration dropdown in the main toolbar, a dialog appears that shows the default CMake configurations available in Visual Studio:

- x86 Debug
- x86 Release
- x64 Debug
- x64 Release
- Linux-Debug
- Linux-Release
- IoT Debug
- IoT Release
- MinGW Debug
- MinGW Release

When you choose a configuration, it's added to the `CMakeSettings.json` file in the project's root folder. You can then use it to build your project. For information about the configuration properties, see CMakeSettings reference.

Linux predefined build configurations:

JSON

```
{
      "name": "Linux-Debug",
      "generator": "Unix Makefiles",
      "remoteMachineName": "user@host",
      "configurationType": "Debug",
      "remoteCMakeListsRoot": "/var/tmp/src/${workspaceHash}/${name}",
      "cmakeExecutable": "/usr/local/bin/cmake",
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "remoteBuildRoot": "/var/tmp/build/${workspaceHash}/build/${name}",
      "remoteInstallRoot":
"/var/tmp/build/${workspaceHash}/install/${name}",
      "remoteCopySources": true,
      "remoteCopySourcesOutputVerbosity": "Normal",
      "remoteCopySourcesConcurrentCopies": "10",
      "remoteCopySourcesMethod": "rsync",
```

```
      "remoteCopySourcesExclusionList": [
        ".vs",
        ".git"
      ],
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [
        "linux_x64"
      ]
}

{
      "name": "Linux-Release",
      "generator": "Unix Makefiles",
      "remoteMachineName": "${defaultRemoteMachineName}",
      "configurationType": "RelWithDebInfo",
      "remoteCMakeListsRoot": "/var/tmp/src/${workspaceHash}/${name}",
      "cmakeExecutable": "/usr/local/bin/cmake",
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "remoteBuildRoot": "/var/tmp/build/${workspaceHash}/build/${name}",
      "remoteInstallRoot":
"/var/tmp/build/${workspaceHash}/install/${name}",
      "remoteCopySources": true,
      "remoteCopySourcesOutputVerbosity": "Normal",
      "remoteCopySourcesConcurrentCopies": "10",
      "remoteCopySourcesMethod": "rsync",
      "remoteCopySourcesExclusionList": [
        ".vs",
        ".git"
      ],
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [
        "linux_x64"
      ]
},
```

You can use these optional settings for more control:

JSON

```
{
      "remotePrebuildCommand": "",
      "remotePreGenerateCommand": "",
```

```
      "remotePostbuildCommand": "",
}
```

These options allow you to run commands on the remote system before and after building, and before CMake generation. The values can be any command that is valid on the remote system. The output is piped back to Visual Studio.

IoT predefined build configurations

JSON

```
{
      "name": "IoT-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "inheritEnvironments": [
        "gcc-arm"
      ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "intelliSenseMode": "linux-gcc-arm",
      "variables": [
        {
          "name": "CMAKE_C_COMPILER",
          "value": "arm-none-eabi-gcc.exe"
        },
        {
          "name": "CMAKE_CXX_COMPILER",
          "value": "arm-none-eabi-g++.exe"
        },
        {
          "name": "CMAKE_C_FLAGS",
          "value": "-nostartfiles"
        },
        {
          "name": "CMAKE_CXX_FLAGS",
          "value": "-nostartfiles -fno-rtti -fno-exceptions"
        },
        {
          "name": "CMAKE_CXX_STANDARD",
          "value": "14"
        },
        {
          "name": "CMAKE_SYSTEM_NAME",
          "value": "Generic"
        },
```

```
        {
          "name": "CMAKE_SYSTEM_PROCESSOR",
          "value": "arm"
        }
      ]
    },
    {
      "name": "IoT-Release",
      "generator": "Ninja",
      "configurationType": "Release",
      "inheritEnvironments": [
        "gcc-arm"
      ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "intelliSenseMode": "linux-gcc-arm",
      "variables": [
        {
          "name": "CMAKE_C_COMPILER",
          "value": "arm-none-eabi-gcc.exe"
        },
        {
          "name": "CMAKE_CXX_COMPILER",
          "value": "arm-none-eabi-g++.exe"
        },
        {
          "name": "CMAKE_C_FLAGS",
          "value": "-nostartfiles"
        },
        {
          "name": "CMAKE_CXX_FLAGS",
          "value": "-nostartfiles -fno-rtti -fno-exceptions"
        },
        {
          "name": "CMAKE_CXX_STANDARD",
          "value": "14"
        },
        {
          "name": "CMAKE_SYSTEM_NAME",
          "value": "Generic"
        },
        {
          "name": "CMAKE_SYSTEM_PROCESSOR",
          "value": "arm"
        }
      ]
    }
```

MinGW predefined build configurations

JSON

```
{
      "environments": [
        {
          "MINGW64_ROOT": "C:\\msys64\\mingw64",
          "BIN_ROOT": "${env.MINGW64_ROOT}\\bin",
          "FLAVOR": "x86_64-w64-mingw32",
          "TOOLSET_VERSION": "7.3.0",
          "PATH":
"${env.MINGW64_ROOT}\\bin;${env.MINGW64_ROOT}\\..\\usr\\local\\bin;${env.MIN
GW64_ROOT}\\..\\usr\\bin;${env.MINGW64_ROOT}\\..\\bin;${env.PATH}",
          "INCLUDE":
"${env.INCLUDE};${env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION};${
env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION}\\tr1;${env.MINGW64_R
OOT}\\include\\c++\\${env.TOOLSET_VERSION}\\${env.FLAVOR}",
          "environment": "mingw_64"
        }
      ],
      "name": "Mingw64-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "inheritEnvironments": [
        "mingw_64"
      ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "intelliSenseMode": "linux-gcc-x64",
      "variables": [
        {
          "name": "CMAKE_C_COMPILER",
          "value": "${env.BIN_ROOT}\\gcc.exe"
        },
        {
          "name": "CMAKE_CXX_COMPILER",
          "value": "${env.BIN_ROOT}\\g++.exe"
        }
      ]
    }

{
      "environments": [
        {
          "MINGW64_ROOT": "C:\\msys64\\mingw64",
          "BIN_ROOT": "${env.MINGW64_ROOT}\\bin",
          "FLAVOR": "x86_64-w64-mingw32",
```

```
          "TOOLSET_VERSION": "7.3.0",
          "PATH":
"${env.MINGW64_ROOT}\\bin;${env.MINGW64_ROOT}\\..\\usr\\local\\bin;${env.MIN
GW64_ROOT}\\..\\usr\\bin;${env.MINGW64_ROOT}\\..\\bin;${env.PATH}",
          "INCLUDE":
"${env.INCLUDE};${env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION};${
env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION}\\tr1;${env.MINGW64_R
OOT}\\include\\c++\\${env.TOOLSET_VERSION}\\${env.FLAVOR}",
          "environment": "mingw_64"
        }
      ],
      "name": "Mingw64-Release",
      "generator": "Ninja",
      "configurationType": "RelWithDebInfo",
      "inheritEnvironments": [
        "mingw_64"
      ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "intelliSenseMode": "linux-gcc-x64",
      "variables": [
        {
          "name": "CMAKE_C_COMPILER",
          "value": "${env.BIN_ROOT}\\gcc.exe"
        },
        {
          "name": "CMAKE_CXX_COMPILER",
          "value": "${env.BIN_ROOT}\\g++.exe"
        }
      ]
    }
```

x86-64 predefined build configurations

JSON

```
    {
      "name": "x86-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "inheritEnvironments": [
        "msvc_x86"
      ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
```

```
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": ""
    },
    {
      "name": "x86-Release",
      "generator": "Ninja",
      "configurationType": "RelWithDebInfo",
      "inheritEnvironments": [
        "msvc_x86"
      ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": ""
    },
    {
      "name": "x64-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "inheritEnvironments": [
        "msvc_x64_x64"
      ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": ""
    },
    {
      "name": "x64-Release",
      "generator": "Ninja",
      "configurationType": "RelWithDebInfo",
      "inheritEnvironments": [
        "msvc_x64_x64"
      ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": ""
    },
    {
      "name": "x86-Release2",
      "generator": "Ninja",
      "configurationType": "RelWithDebInfo",
      "inheritEnvironments": [
        "msvc_x86"
```

```
      ],
      "buildRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
      "installRoot":
"${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": ""
    }
  ]
}
```

In a CMake project, build configurations are stored in a `CMakeSettings.json` file. When you choose **Manage Configurations** from the build configuration dropdown in the main toolbar, a dialog appears that shows the default CMake configurations available in Visual Studio:

- x86 Debug
- x86 Clang Debug
- x86 Release
- x86 Clang Release
- x64 Debug
- x64 Clang Debug
- x64 Release
- x64 Clang Release
- Linux-Debug
- Linux-Release
- Linux-Clang-Debug
- Linux-Clang-Release
- Existing Cache (remote)
- Existing Cache
- MinGW Debug
- MinGW Release
- WSL Debug
- WSL Release
- WSL Clang Debug
- WSL Clang Release
- Clang

When you choose a configuration, it's added to the `CMakeSettings.json` file in the project's root folder. You can then use it to build your project.

JSON

```
{
  "configurations": [
    {
      "name": "x64-Debug",
      "generator": "Ninja",
      "configurationType": "Release",
      "inheritEnvironments": [ "msvc_x64_x64" ],
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "variables": []
    },
    {
      "name": "Linux-Debug",
      "generator": "Unix Makefiles",
      "configurationType": "Release",
      "cmakeExecutable": "/usr/bin/cmake",
      "remoteCopySourcesExclusionList": [ ".vs", ".git", "out" ],
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_x64" ],
      "remoteMachineName": "${defaultRemoteMachineName}",
      "remoteCMakeListsRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/src",
      "remoteBuildRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/out/build/${name}",
      "remoteInstallRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/out/install/${name}",
      "remoteCopySources": true,
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "remoteCopySourcesMethod": "rsync",
      "addressSanitizerRuntimeFlags": "detect_leaks=0",
      "variables": []
    },
    {
      "name": "Linux-Release",
      "generator": "Unix Makefiles",
      "configurationType": "RelWithDebInfo",
      "cmakeExecutable": "/usr/bin/cmake",
      "remoteCopySourcesExclusionList": [ ".vs", ".git", "out" ],
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_x64" ],
      "remoteMachineName": "${defaultRemoteMachineName}",
      "remoteCMakeListsRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/src",
      "remoteBuildRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/out/build/${name}",
      "remoteInstallRoot":
```

```
"$HOME/.vs/${projectDirName}/${workspaceHash}/out/install/${name}",
      "remoteCopySources": true,
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "remoteCopySourcesMethod": "rsync",
      "addressSanitizerRuntimeFlags": "detect_leaks=0",
      "variables": []
    },
    {
      "name": "Linux-Clang-Debug",
      "generator": "Unix Makefiles",
      "configurationType": "Debug",
      "cmakeExecutable": "/usr/bin/cmake",
      "remoteCopySourcesExclusionList": [ ".vs", ".git", "out" ],
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_clang_x64" ],
      "remoteMachineName": "${defaultRemoteMachineName}",
      "remoteCMakeListsRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/src",
      "remoteBuildRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/out/build/${name}",
      "remoteInstallRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/out/install/${name}",
      "remoteCopySources": true,
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "remoteCopySourcesMethod": "rsync",
      "addressSanitizerRuntimeFlags": "detect_leaks=0",
      "variables": []
    },
    {
      "name": "Linux-Clang-Release",
      "generator": "Unix Makefiles",
      "configurationType": "RelWithDebInfo",
      "cmakeExecutable": "/usr/bin/cmake",
      "remoteCopySourcesExclusionList": [ ".vs", ".git", "out" ],
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_clang_x64" ],
      "remoteMachineName": "${defaultRemoteMachineName}",
      "remoteCMakeListsRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/src",
      "remoteBuildRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/out/build/${name}",
      "remoteInstallRoot":
"$HOME/.vs/${projectDirName}/${workspaceHash}/out/install/${name}",
      "remoteCopySources": true,
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "remoteCopySourcesMethod": "rsync",
      "addressSanitizerRuntimeFlags": "detect_leaks=0",
      "variables": []
```

```
    },
    {
      "name": "Existing Cache (Remote)",
      "cacheRoot": "",
      "remoteCopySourcesExclusionList": [ ".vs", ".git", "out" ],
      "inheritEnvironments": [ "linux_x64" ],
      "remoteMachineName": "${defaultRemoteMachineName}",
      "remoteCopySources": false,
      "rsyncCommandArgs": "-t --delete --delete-excluded",
      "remoteCopyBuildOutput": false,
      "remoteCopySourcesMethod": "rsync",
      "addressSanitizerRuntimeFlags": "detect_leaks=0",
      "variables": []
    },
    {
      "name": "Mingw64-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir\\out\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "mingw_64" ],
      "environments": [
        {
          "MINGW64_ROOT": "C:\\msys64\\mingw64",
          "BIN_ROOT": "${env.MINGW64_ROOT}\\bin",
          "FLAVOR": "x86_64-w64-mingw32",
          "TOOLSET_VERSION": "7.3.0",
          "PATH":
"${env.MINGW64_ROOT}\\bin;${env.MINGW64_ROOT}\\..\\usr\\local\\bin;${env.MIN
GW64_ROOT}\\..\\usr\\bin;${env.MINGW64_ROOT}\\..\\bin;${env.PATH}",
          "INCLUDE":
"${env.INCLUDE};${env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION};${
env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION}\\tr1;${env.MINGW64_R
OOT}\\include\\c++\\${env.TOOLSET_VERSION}\\${env.FLAVOR}",
          "environment": "mingw_64"
        }
      ],
      "variables": [
        {
          "name": "CMAKE_C_COMPILER",
          "value": "${env.BIN_ROOT}\\gcc.exe",
          "type": "STRING"
        },
        {
          "name": "CMAKE_CXX_COMPILER",
          "value": "${env.BIN_ROOT}\\g++.exe",
          "type": "STRING"
        }
      ],
      "intelliSenseMode": "linux-gcc-x64"
    },
    {
```

```
      "name": "Mingw64-Release",
      "generator": "Ninja",
      "configurationType": "RelWithDebInfo",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir\\out\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "mingw_64" ],
      "environments": [
        {
          "MINGW64_ROOT": "C:\\msys64\\mingw64",
          "BIN_ROOT": "${env.MINGW64_ROOT}\\bin",
          "FLAVOR": "x86_64-w64-mingw32",
          "TOOLSET_VERSION": "7.3.0",
          "PATH":
"${env.MINGW64_ROOT}\\bin;${env.MINGW64_ROOT}\\..\\usr\\local\\bin;${env.MIN
GW64_ROOT}\\..\\usr\\bin;${env.MINGW64_ROOT}\\..\\bin;${env.PATH}",
          "INCLUDE":
"${env.INCLUDE};${env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION};${
env.MINGW64_ROOT}\\include\\c++\\${env.TOOLSET_VERSION}\\tr1;${env.MINGW64_R
OOT}\\include\\c++\\${env.TOOLSET_VERSION}\\${env.FLAVOR}",
          "environment": "mingw_64"
        }
      ],
      "variables": [
        {
          "name": "CMAKE_C_COMPILER",
          "value": "${env.BIN_ROOT}\\gcc.exe",
          "type": "STRING"
        },
        {
          "name": "CMAKE_CXX_COMPILER",
          "value": "${env.BIN_ROOT}\\g++.exe",
          "type": "STRING"
        }
      ],
      "intelliSenseMode": "linux-gcc-x64"
    },
    {
      "name": "x64-Release",
      "generator": "Ninja",
      "configurationType": "RelWithDebInfo",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "msvc_x64_x64" ],
      "variables": []
    },
    {
      "name": "x86-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
```

```
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "msvc_x86" ],
      "variables": []
    },
    {
      "name": "x86-Release",
      "generator": "Ninja",
      "configurationType": "RelWithDebInfo",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "msvc_x86" ],
      "variables": []
    },
    {
      "name": "x86-Clang-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "clang_cl_x86" ],
      "variables": []
    },
    {
      "name": "x86-Clang-Release",
      "generator": "Ninja",
      "configurationType": "RelWithDebInfo",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "-v",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "clang_cl_x86" ],
      "variables": []
    },
    {
      "name": "Existing Cache",
      "cacheRoot": "",
      "inheritEnvironments": [],
      "variables": []
    },
    {
      "name": "WSL-Debug",
      "generator": "Unix Makefiles",
      "configurationType": "Debug",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
```

```
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeExecutable": "/usr/bin/cmake",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_x64" ],
      "wslPath": "${defaultWSLPath}",
      "addressSanitizerRuntimeFlags": "detect_leaks=0",
      "variables": []
    },
    {
      "name": "WSL-Release",
      "generator": "Unix Makefiles",
      "configurationType": "RelWithDebInfo",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeExecutable": "/usr/bin/cmake",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_x64" ],
      "wslPath": "${defaultWSLPath}",
      "addressSanitizerRuntimeFlags": "detect_leaks=0",
      "variables": []
    },
    {
      "name": "WSL-Clang-Debug",
      "generator": "Unix Makefiles",
      "configurationType": "Debug",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeExecutable": "/usr/bin/cmake",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_clang_x64" ],
      "wslPath": "${defaultWSLPath}",
      "addressSanitizerRuntimeFlags": "detect_leaks=0",
      "variables": []
    },
    {
      "name": "WSL-Clang-Release",
      "generator": "Unix Makefiles",
      "configurationType": "RelWithDebInfo",
      "buildRoot": "${projectDir}\\out\\build\\${name}",
      "installRoot": "${projectDir}\\out\\install\\${name}",
      "cmakeExecutable": "/usr/bin/cmake",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": "",
      "inheritEnvironments": [ "linux_clang_x64" ],
      "wslPath": "${defaultWSLPath}",
      "addressSanitizerRuntimeFlags": "detect_leaks=0",
      "variables": []
    }
```

```
  ]
}
```

See also

CMake Projects in Visual Studio
Configure a Linux CMake project
Connect to your remote Linux computer
Configure CMake debugging sessions
Deploy, run, and debug your Linux project
CMake predefined configuration reference

Get started with C++ Build Insights

Article • 11/12/2024

C++ Build Insights is a collection of tools that collect data about your C++ builds, and present it in a format that can help you answer common questions such as:

- Are my builds sufficiently parallelized?
- What should I include in my pre-compiled header (PCH)?
- Is there a specific bottleneck I should focus on to increase my build speeds?

The main components of this technology are:

- `vcperf.exe`, a command-line utility that you can use to collect traces for your builds
- A Windows Performance Analyzer (WPA) extension that allows you to view build traces in WPA, and
- The C++ Build Insights software development kit for creating your own tools that consume C++ Build Insights data.

Documentation sections

vcperf and Windows Performance Analyzer
Learn how to collect build traces for your C++ projects and how to view them in WPA.

Windows Performance Basics
Discover useful WPA tips for analyzing your build traces.

C++ Build Insights SDK
An overview of the C++ Build Insights SDK.

Articles

Read these articles from the official C++ team blog for more information on C++ Build Insights:

Introducing C++ Build Insights
Analyze your builds programmatically with the C++ Build Insights SDK
Finding build bottlenecks with C++ Build Insights
Faster builds with PCH suggestions from C++ Build Insights
Profiling template metaprograms with C++ Build Insights
Improving code generation time with C++ Build Insights

Introducing vcperf /timetrace for C++ build time analysis

Faster C++ builds, simplified: a new metric for time

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Compare header units, modules, and precompiled headers

Article • 12/05/2022

Historically, you'd include the standard library with a directive like `#include <vector>`. However, it's expensive to include header files because they're reprocessed by every source file that includes them.

Precompiled headers (PCH) were introduced to speed compilation by translating them once and reusing the result. But precompiled headers can be difficult to maintain.

In C++20, modules were introduced as a significant improvement on header files and precompiled headers.

Header units were introduced in C++20 as a way to temporarily bridge the gap between header files and modules. They provide some of the speed and robustness benefits of modules, while you migrate your code to use modules.

Then, the C++23 standard library introduced support for importing the standard library as named modules. This is the fastest and most robust way to consume the standard library.

To help you sort out the different options, this article compares the traditional `#include` method against precompiled headers, header units, and importing named modules.

The following table is arranged by compiler processing speed and robustness, with `#include` being the slowest and least robust, and `import` being the fastest and most robust.

Method	Summary
`#include`	One disadvantage is that they expose macros and internal implementation. Internal implementation is often exposed as functions and types that start with an underscore. That's a convention to indicate that something is part of the internal implementation and shouldn't be used. Header files are fragile because the order of #includes can modify behavior or break code and are affected by macro definitions. Header files slow compilation. Particularly when multiple files include the same file because then the header file is reprocessed multiple times.

Method	Summary
Precompiled header	A precompiled header (PCH) improves compile time by creating a compiler memory snapshot of a set of header files. This is an improvement on repeatedly rebuilding header files. PCH files have restrictions that make them difficult to maintain. PCH files are faster than `#include` but slower than `import`.
Header units	This is a new feature in C++20 that allows you to import 'well-behaved' header files as modules. Header units are faster than `#include`, and are easier to maintain, significantly smaller, and also faster than pre-compiled header files (PCH). Header units are an 'in-between' step meant to help transition to named modules in cases where you rely on macros defined in header files, since named modules don't expose macros. Header units are slower than importing a named module. Header units aren't affected by macro defines unless they're specified on the command line when the header unit is built--making them more robust than header files. Header units expose the macros and internal implementation defined in them just as header file do, which named modules don't. As a rough approximation of file size, a 250-megabyte PCH file might be represented by an 80-megabyte header unit file.

Method	Summary
Modules	This is the fastest and most robust way to import functionality. Support for importing modules was introduced in C++20. The C++23 standard library introduces the two named modules described in this topic. When you import `std`, you get the standard names such as `std::vector`, `std::cout`, but no extensions, no internal helpers such as `_Sort_unchecked`, and no macros. The order of imports doesn't matter because there are no macro or other side-effects. As a rough approximation of file size, a 250-megabyte PCH file might be represented by an 80-megabyte header unit file, which might be represented by a 25-megabyte module. Named modules are faster because when a named module is compiled into an `.ifc` file and an `.obj` file, the compiler emits a structured representation of the source code that can be loaded quickly when the module is imported. The compiler can do some work (like name resolution) before emitting the `.ifc` file because of how named modules are order-independent and macro-independent--so this work doesn't have to be done when the module is imported. In contrast, when a header file is consumed with `#include`, its contents must be preprocessed and compiled again and again in every translation unit. Precompiled headers, which are compiler memory snapshots, can mitigate those costs, but not as well as named modules.

If you can use C++20 features and the C++23 standard library in your app, use named modules.

If you can use C++20 features but want to transition over time to modules, use header units in the interim.

If you can't use C++20 features, use `#include` and consider precompiled headers.

See also

Precompiled header files
Overview of modules in C++
Tutorial: Import the C++ standard library using modules
Walkthrough: Import STL libraries as header units
Walkthrough: Build and import header units in your Visual C++ projects

Walkthrough: Build and import header units in Microsoft Visual C++

Article • 12/05/2022

This article is about building and importing header units with Visual Studio 2022. To learn how to import C++ standard library headers as header units, see Walkthrough: Import STL libraries as header units. For an even faster and more robust way to import the standard library, see Tutorial: Import the C++ standard library using modules.

Header units are the recommended alternative to precompiled header files (PCH). Header units are easier to set up and use, are significantly smaller on disk, provide similar performance benefits, and are more flexible than a shared PCH.

To contrast header units with other ways to include functionality in your programs, see Compare header units, modules, and precompiled headers.

Prerequisites

To use header units, you need Visual Studio 2019 16.10 or later.

What is a header unit

A header unit is a binary representation of a header file. A header unit ends with an `.ifc` extension. The same format is used for named modules.

An important difference between a header unit and a header file is that a header unit isn't affected by macro definitions outside of the header unit. That is, you can't define a preprocessor symbol that causes the header unit to behave differently. By the time you import the header unit, the header unit is already compiled. That's different from how an `#include` file is treated. An included file can be affected by a macro definition outside of the header file because the header file goes through the preprocessor when you compile the source file that includes it.

Header units can be imported in any order, which isn't true of header files. Header file order matters because macro definitions defined in one header file might affect a subsequent header file. Macro definitions in one header unit can't affect another header unit.

Everything visible from a header file is also visible from a header unit, including macros defined within the header unit.

A header file must be translated into a header unit before it can be imported. An advantage of header units over precompiled header files (PCH) is that they can be used in distributed builds. As long as you compile the `.ifc` and the program that imports it with the same compiler, and target the same platform and architecture, a header unit produced on one computer can be consumed on another. Unlike a PCH, when a header unit changes, only it and what depends on it are rebuilt. Header units can be up to an order of magnitude smaller in size than a `.pch`.

Header units impose fewer constraints on the required similarities of compiler switch combinations used to create the header unit and to compile the code that consumes it than a PCH does. However, some switch combinations and macro definitions might create violations of the one definition rule (ODR) between various translation units.

Finally, header units are more flexible than a PCH. With a PCH, you can't choose to bring in only one of the headers in the PCH--the compiler processes all of them. With header units, even when you compile them together into a static library, you only bring the contents of the header unit you import into your application.

Header units are a step in between header files and C++ 20 modules. They provide some of the benefits of modules. They're more robust because outside macro definitions don't affect them--so you can import them in any order. And the compiler can process them faster than header files. But header units don't have all of the advantages of modules because header units expose the macros defined within them (modules don't). Unlike modules, there's no way to hide private implementation in a header unit. To indicate private implementation with header files, different techniques are employed like adding leading underscores to names, or putting things in an implementation namespace. A module doesn't expose private implementation in any form, so you don't need to do that.

Consider replacing your precompiled headers with header units. You get the same speed advantage, but with other code hygiene and flexibility benefits as well.

Ways to compile a header unit

There are several ways to compile a file into a header unit:

- **Build a shared header unit project**. We recommend this approach because it provides more control over the organization and reuse of the imported header units. Create a static library project that contains the header units that you want, and then reference it to import the header units. For a walkthrough of this approach, see Build a header unit static library project for header units.

- **Choose individual files to translate into header units**. This approach gives you file-by-file control over what is treated as a header unit. It's also useful when you must compile a file as a header unit that, because it doesn't have the default extension (`.ixx`, `.cppm`, `.h`, `.hpp`), wouldn't normally be compiled into a header unit. This approach is demonstrated in this walkthrough. To get started, see Approach 1: Translate a specific file into a header unit.

- **Automatically scan for and build header units**. This approach is convenient, but is best suited to smaller projects because it doesn't guarantee optimal build throughput. For details about this approach, see Approach 2: Automatically scan for header units.

- As mentioned in the introduction, you can build and import STL header files as header units, and automatically treat `#include` for STL library headers as `import` without rewriting your code. To see how, visit Walkthrough: Import STL libraries as header units.

Approach 1: Translate a specific file into a header unit

This section shows how to choose a specific file to translate into a header unit. Compile a header file as a header unit using the following steps in Visual Studio:

1. Create a new C++ console app project.

2. Replace the source file content as follows:

```
C++
```

```
#include "Pythagorean.h"

int main()
{
    PrintPythagoreanTriple(2,3);
    return 0;
}
```

3. Add a header file called `Pythagorean.h` and then replace its content with this code:

```
C++
```

```
#ifndef PYTHAGOREAN
#define PYTHAGOREAN
```

```
#include <iostream>

inline void PrintPythagoreanTriple(int a, int b)
{
    std::cout << "Pythagorean triple a:" << a << " b:" << b << " c:" <<
a*a + b*b << std::endl;
}
#endif
```

Set project properties

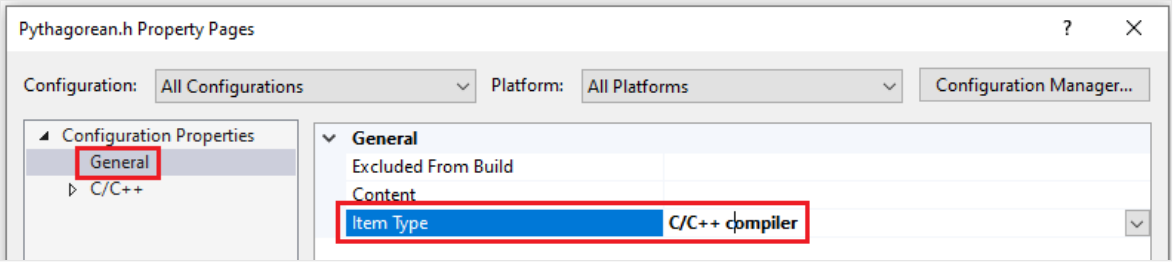
To enable header units, first set the **C++ Language Standard** to /std:c++20 or later with the following steps:

1. In **Solution Explorer**, right-click the project name and choose **Properties**.
2. In the left pane of the project property pages window, select **Configuration Properties** > **General**.
3. In the **C++ Language Standard** dropdown, select **ISO C++20 Standard (/std:c++20)** or later. Choose **Ok** to close the dialog.

Compile the header file as a header unit:

1. In **Solution Explorer**, select the file you want to compile as a header unit (in this case, `Pythagorean.h`). Right-click the file and choose **Properties**.

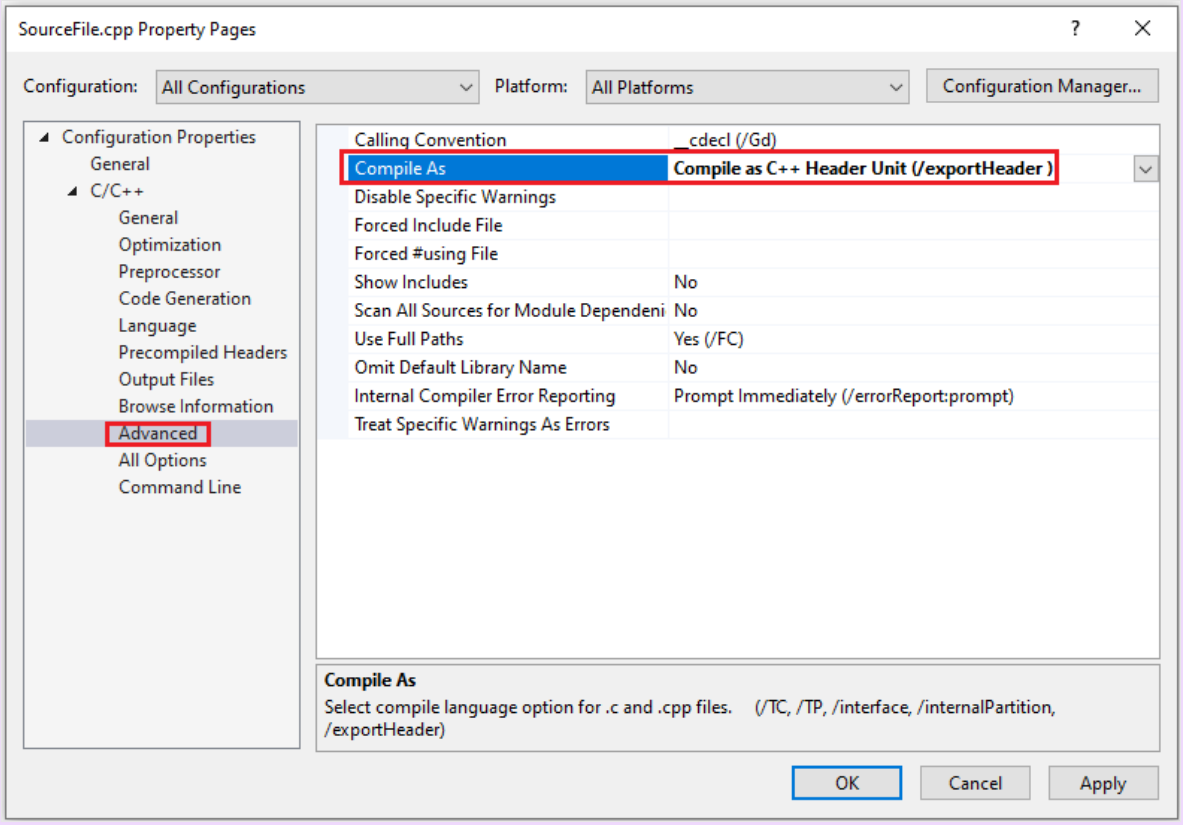
2. Set the **Configuration properties** > **General** > **Item Type** dropdown to **C/C++ compiler** and choose **Ok**.



When you build this project later in this walkthrough, `Pythagorean.h` will be translated into a header unit. It's translated into a header unit because the item type for this header file is set to **C/C++ compiler**, and because the default action for `.h` and `.hpp` files set this way is to translate the file into a header unit.

> **Note**
>
> This isn't required for this walkthrough, but is provided for your information. To compile a file as a header unit that doesn't have a default header unit file extension,

like `.cpp` for example, set **Configuration properties** > **C/C++** > **Advanced** > **Compile As** to **Compile as C++ Header Unit (/exportHeader)**:



Change your code to import the header unit

1. In the source file for the example project, change `#include "Pythagorean.h"` to `import "Pythagorean.h";` Don't forget the trailing semicolon. It's required for `import` statements. Because it's a header file in a directory local to the project, we used quotes with the `import` statement: `import "file";`. In your own projects, to compile a header unit from a system header, use angle brackets: `import <file>;`

2. Build the solution by selecting **Build** > **Build Solution** on the main menu. Run it to see that it produces the expected output: `Pythagorean triple a:2 b:3 c:13`

In your own projects, repeat this process to compile the header files you want to import as header units.

If you want to convert only a few header files to header units, this approach is good. But if you have many header files that you want to compile, and the potential loss of build performance is outweighed by the convenience of having the build system handle them automatically, see the following section.

If you're interested in specifically importing STL library headers as header units, see Walkthrough: Import STL libraries as header units.

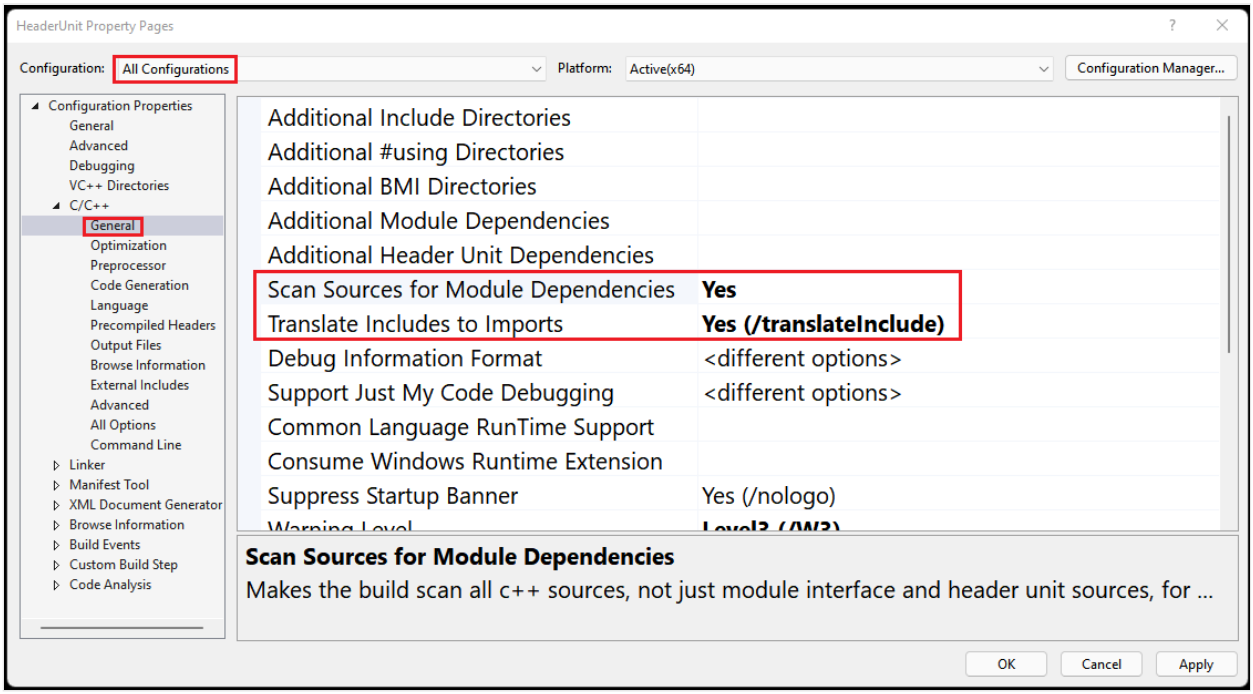
Approach 2: Automatically scan for and build header units

Because it takes time to scan all of your source files for header units, and time to build them, the following approach is best suited for smaller projects. It doesn't guarantee optimal build throughput.

This approach combines two Visual Studio project settings:

- **Scan Sources for Module Dependencies** causes the build system to call the compiler to ensure that all imported modules and header units are built before compiling the files that depend on them. When combined with **Translate Includes to Imports**, any header files included in your source that are also specified in a header-units.json file located in the same directory as the header file, are compiled into header units.
- **Translate Includes to Imports** treats a header file as an `import` if the `#include` refers to a header file that can be compiled as a header unit (as specified in a `header-units.json` file), and a compiled header unit is available for the header file. Otherwise, the header file is treated as a normal `#include`. The header-units.json file is used to automatically build header units for each `#include`, without symbol duplication.

You can turn on these settings in the properties for your project. To do so, right-click the project in the **Solution Explorer** and choose **Properties**. Then choose **Configuration Properties** > **C/C++** > **General**.



Scan Sources for Module Dependencies can be set for all of the files in the project in **Project Properties** as shown here, or for individual files in **File Properties**. Modules and header units are always scanned. Set this option when you have a `.cpp` file that imports header units that you want built automatically and might not be built yet.

These settings work together to automatically build and import header units under these conditions:

- **Scan Sources for Module Dependencies** scans your sources for the files and their dependencies that can be treated as header units. Files that have the extension `.ixx`, and files that have their **File properties** > **C/C++** > **Compile As** property set to **Compile as C++ Header Unit (/export)**, are always scanned regardless of this setting. The compiler also looks for `import` statements to identify header unit dependencies. If `/translateInclude` is specified, the compiler also scans for `#include` directives that are also specified in a `header-units.json` file to treat as header units. A dependency graph is built of all the modules and header units in your project.
- **Translate Includes to Imports** When the compiler encounters an `#include` statement, and a matching header unit file (`.ifc`) exists for the specified header file, the compiler imports the header unit instead of treating the header file as an `#include`. When combined with **Scan for dependencies**, the compiler finds all of the header files that can be compiled into header units. An allowlist is consulted by the compiler to decide which header files can compile into header units. This list is stored in a header-units.json file that must be in the same directory as the included file. You can see an example of a `header-units.json` file under the installation directory for Visual Studio. For example, `%ProgramFiles%\Microsoft Visual Studio\2022\Enterprise\VC\Tools\MSVC\14.30.30705\include\header-units.json` is used by the compiler to determine whether a Standard Template Library header can be compiled into a header unit. This functionality exists to serve as a bridge with legacy code to get some benefits of header units.

The `header-units.json` file serves two purposes. In addition to specifying which header files can be compiled into header units, it minimizes duplicated symbols to increase build throughput. For more information about symbol duplication, see C++ header-units.json reference.

These switches and the `header-unit.json` provide some of the benefits of header units. The convenience comes at the cost of build throughput. This approach might not be the best for larger projects because it doesn't guarantee optimal build times. Also, the same header files might be reprocessed repeatedly, which increases build time. However, the convenience might be worth it depending on the project.

These features are designed for legacy code. For new code, move to modules instead of header units or `#include` files. For a tutorial on using modules, see Name modules tutorial (C++).

For an example of how this technique is used to import STL header files as header units, see Walkthrough: Import STL libraries as header units.

Preprocessor implications

The standard C99/C++11 conforming preprocessor is required to create and use header units. The compiler enables the new C99/C++11 conforming preprocessor when compiling header units by implicitly adding /Zc:preprocessor to the command line whenever any form of `/exportHeader` is used. Attempting to turn it off will result in a compilation error.

Enabling the new preprocessor affects the processing of variadic macros. For more information, see the Variadic macros remarks section.

See also

/translateInclude
/exportHeader
/headerUnit
header-units.json
Compare header units, modules, and precompiled headers
Overview of modules in C++
Tutorial: Import the C++ standard library using modules
Walkthrough: Import STL libraries as header units

Walkthrough: Import STL libraries as header units

Article • 12/05/2022

This walkthrough shows how to import C++ Standard Template Library (STL) libraries as header units in Visual Studio. For an even faster and more robust way to import the standard library, see Tutorial: Import the C++ standard library using modules.

Importing an STL header as a header unit is simpler than using precompiled header files. Header units are easier to set up and use, are substantially smaller on disk, provide similar performance benefits, and are more flexible than a shared PCH.

For more detailed information about what header units are and the benefits they provide, see What is a header unit?. To contrast header units with other ways to import the standard library, see Compare header units, modules, and precompiled headers.

Prerequisites

To use header units, use Visual Studio 2022 or later, or Visual Studio 2019 version 16.11 or later. The /std:c++20 option (or later) is required to use header units.

Two approaches to import STL headers as header units

Before you can import an STL header, it must be compiled into a header unit. A header unit is a binary representation of a header file. It has an `.ifc` extension.

The recommended approach is to create a static library that contains the built header units for the STL headers you want to use. Then reference that library and `import` its header units. This approach can result in faster builds and better reuse. To try out this approach, see Approach 1: Create a static library of STL library header units.

Another approach is to have Visual Studio scan for the STL headers you `#include` in your project, compile them into header units, and `import` rather than `#include` those headers. This approach is useful if you have a large codebase, because you don't have to change your source code. This approach is less flexible than the static library approach, because it doesn't lend itself to reusing the built header units in other projects. But, you still get the performance advantage of importing individual STL libraries as header units. To try out this approach, see Approach 2: Scan includes for STL headers to import.

Approach 1: Create a static library of STL library header units

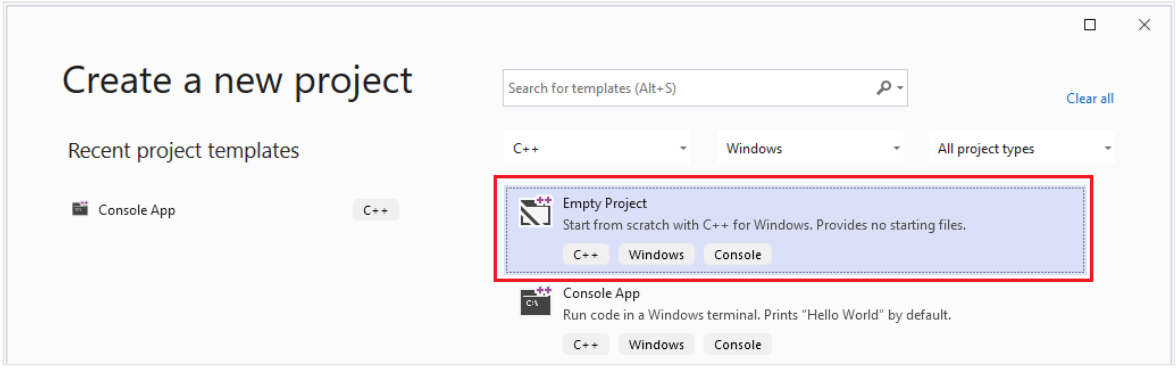
The recommended way to consume STL libraries as header units is to create one or more static library projects. These projects should consist of the STL library header units that you want to use. Then, reference the library projects to consume those STL header units. It's similar to using shared precompiled headers, but easier.

Header units (and modules) built in a static library project are automatically available to referencing projects because the project system automatically adds the appropriate /headerUnit command-line option to the compiler so that referencing projects can import the header units.

This approach ensures that the header unit for a particular header is built only once. It allows you to import some or all of the header units, which isn't possible with a PCH. You can include header units in any order.

In the following example, you create a static library project consisting of the `<iostream>` and `<vector>` header units. After the solution is built, you'll reference this shared header unit project from another C++ project. Everywhere `import <iostream>;` or `import <vector>;` is found, the built header unit for that library is used instead of translating the header with the preprocessor. It improves build performance, like PCH files do, when the same header is included in multiple files. The header won't have to be processed over and over by the files that include it. Instead, the already processed compiled header unit is imported.

To create a static library that contains the STL libraries `<iostream>` and `<vector>`, follow these steps:

1. Create an empty C++ project. Name it *SharedPrj*.
 Select **Empty Project** for C++ from the project types available in the **Create a new project** window:
 

2. Add a new C++ file to the project. Change the file's content to:

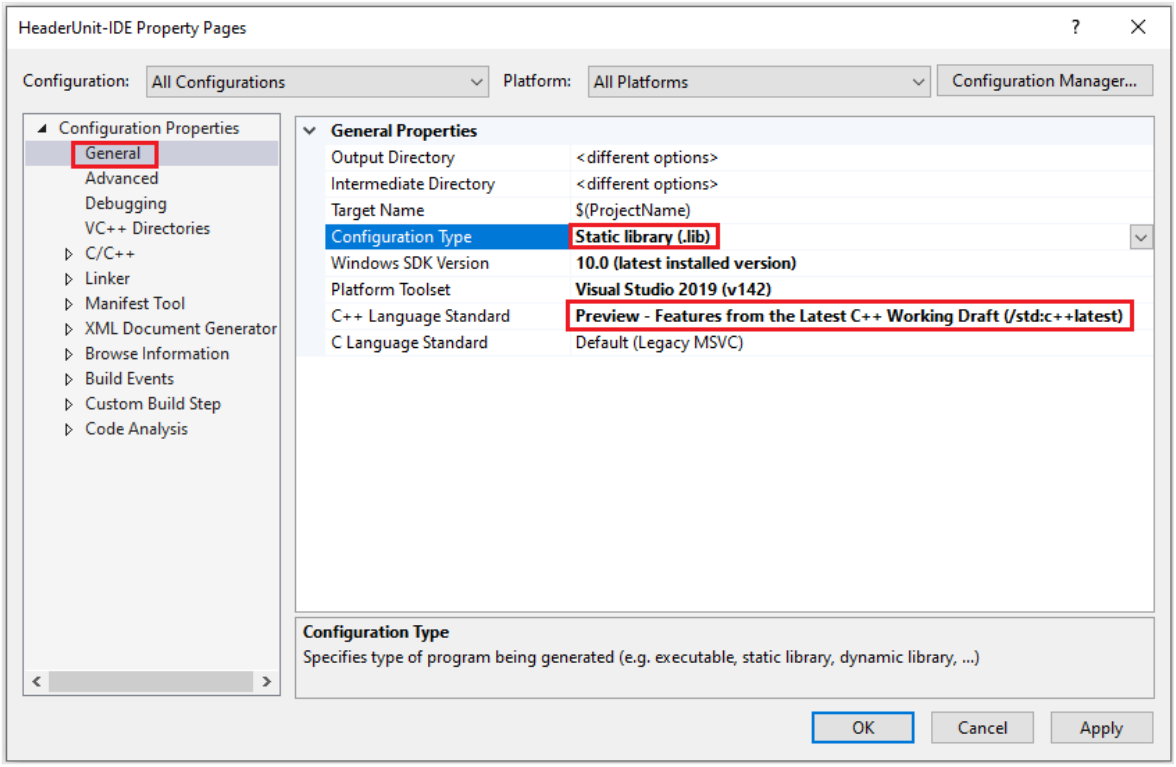
```
C++
```

```
import <iostream>;
import <vector>;
```

Set project properties

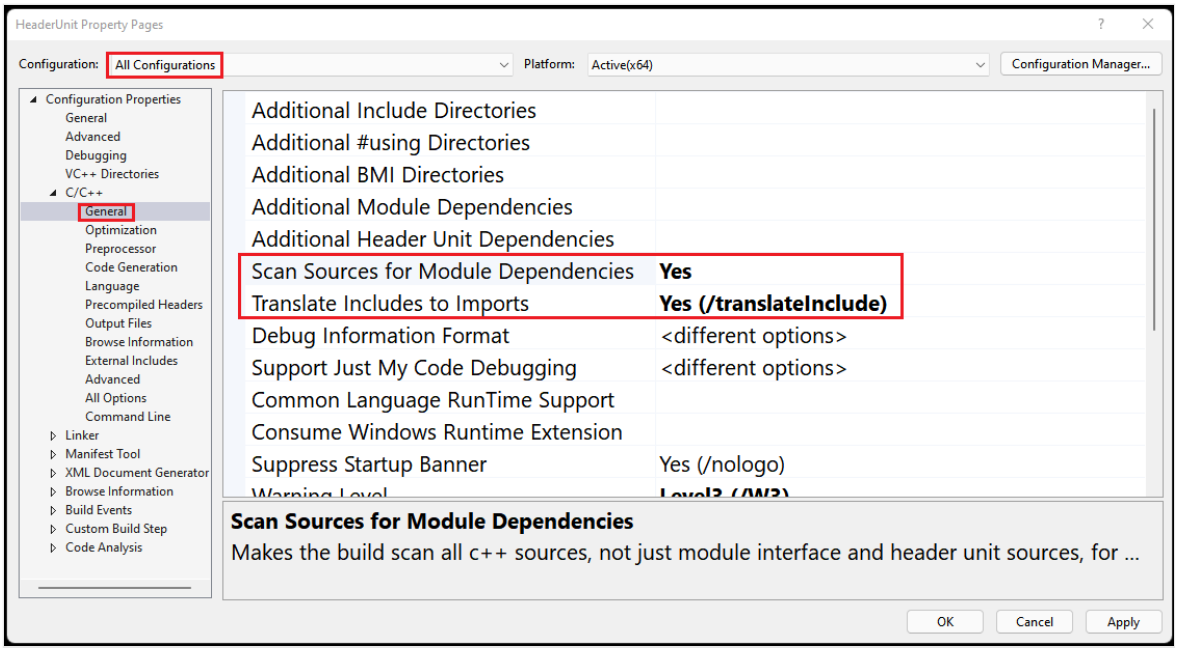
Set project properties to share the header units from this project:

1. On the Visual Studio main menu, select **Project** > **SharedPrj Properties** to open the project Property Pages dialog:



2. Select **All Configurations** in the **Configuration** dropdown list, and then select **All Platforms** in the **Platform** dropdown list. These settings ensure that your changes apply whether you're building for debug or release.
3. In the left pane of the project Property Pages dialog, select **Configuration Properties** > **General**.
4. Change the **Configuration Type** option to **Static library (.lib)**.
5. Change **C++ Language Standard** to **ISO C++20 Standard (/std:c++20)** (or later).
6. In the left pane of the project Property Pages dialog, select **Configuration Properties** > **C/C++** > **General**.
7. In the **Scan Sources for Module Dependencies** dropdown list, select **Yes**. (This option causes the compiler to scan your code for dependencies that can be built

into header units):



8. Choose **OK** to close the project Property Pages dialog. Build the solution by selecting **Build** > **Build Solution** on the main menu.

Reference the header unit library

To import `<iostream>` and `<vector>` as header units from the static library, create a project that references the static library as follows:

1. With the current solution still open, on the Visual Studio menu, select **File** > **Add** > **New Project**.

2. In the **Create a new project** wizard, select the C++ **Console App** template and choose **Next**.

3. Name the new project *Walkthrough*. Change the **Solution** dropdown to **Add to solution**. Choose **Create** to create the project and add it to your solution.

4. Change the content of the *Walkthrough.cpp* source file as follows:

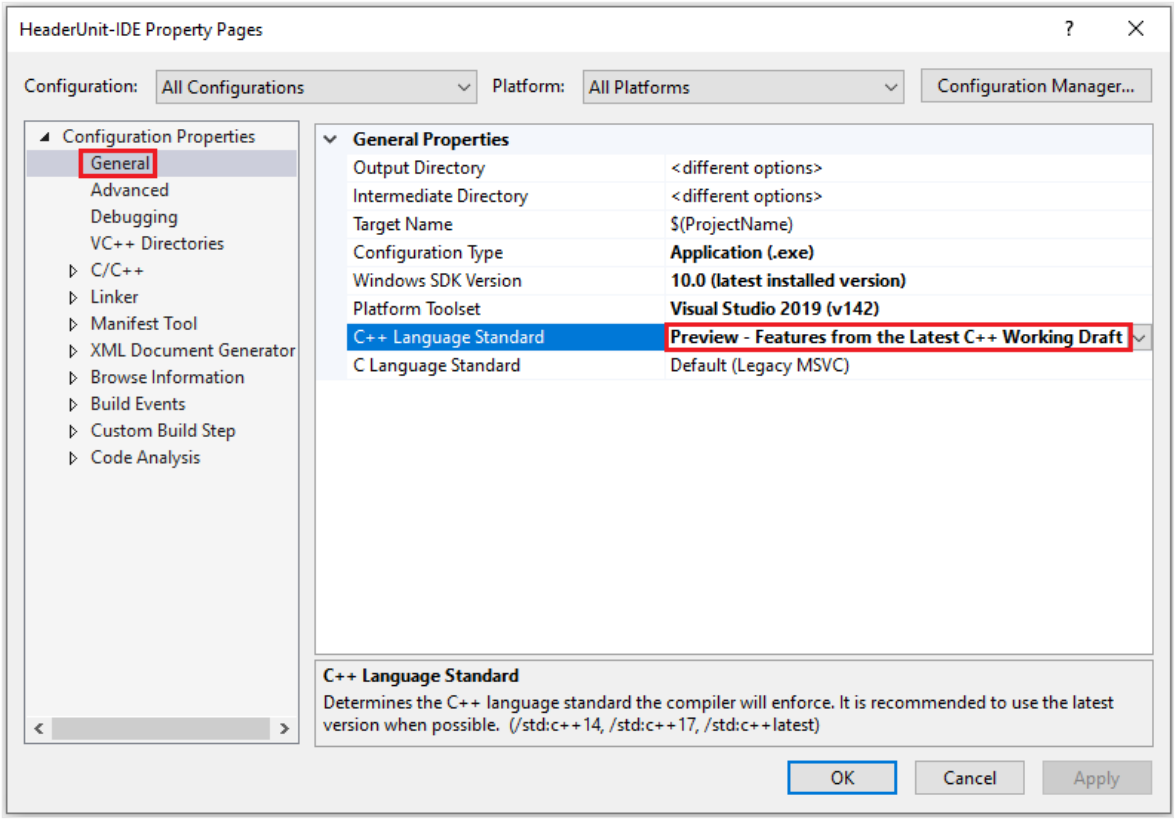
```
C++

import <iostream>;
import <vector>;

int main()
{
    std::vector<int> numbers = {0, 1, 2};
    std::cout << numbers[1];
}
```

Header units require the /std:c++20 option (or later). Set the language standard by using the following steps:

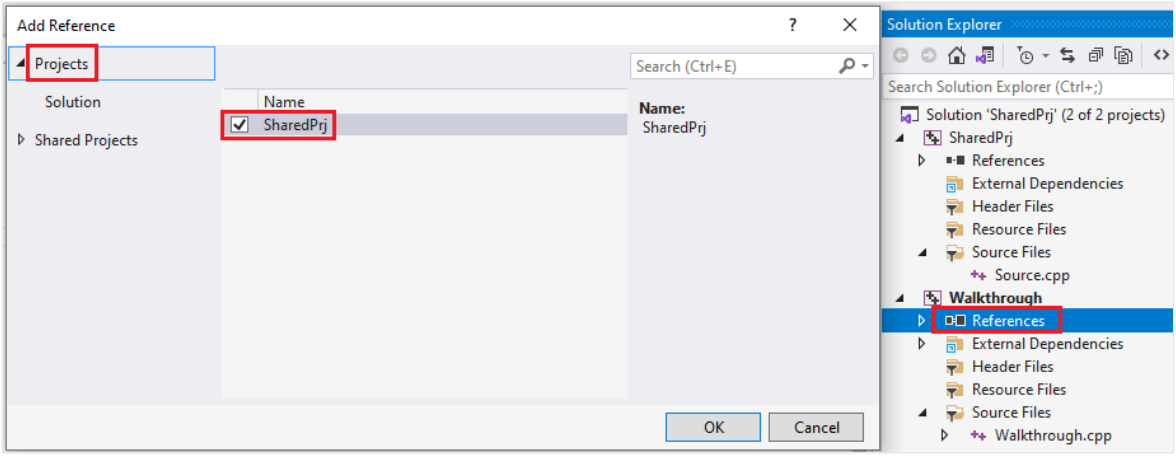
1. In **Solution Explorer**, right-click the **Walkthrough** project and select **Properties** to open the project Property Pages dialog:



2. In the left pane of the **Walkthrough** project Property Pages dialog, select **Configuration Properties** > **General**.
3. In the **C++ Language Standard** dropdown, select **ISO C++20 Standard (/std:c++20)** (or later).
4. Choose **OK** to close the project Property Pages dialog.

In the **Walkthrough** project, add a reference to the **SharedPrj** project with the following steps:

1. In the **Walkthrough** project, select the **References** node, and then select **Add Reference**. Select **SharedPrj** in the list of projects:



Adding this reference causes the build system to use the header units built by **SharedPrj** whenever an `import` in the **Walkthrough** project matches one of the built header units in **SharedPrj**.

2. Choose **OK** to close the **Add Reference** dialog.
3. Right-click the **Walkthrough** project and select **Set as Startup Project**.
4. Build the solution. (Use **Build** > **Build Solution** on the main menu.) Run it to see that it produces the expected output: `1`

The advantage of this approach is that you can reference the static library project from any project to reuse the header units in it. In this example, the static library contains the `<vector>` and `<iostream>` header units.

You can make a monolithic static library project that contains all the commonly used STL headers that you want to import from your various projects. Or you can create smaller shared library projects for the different groupings of STL libraries that you want to import as header units. Then reference those shared header unit projects as needed.

The result should be increased build throughput because importing a header unit significantly reduces the work the compiler must do.

When you use this approach with your own projects, build the static library project with compiler options that are compatible with the project that references it. For example, STL projects should be built with the **`/EHsc`** compiler option to turn on exception handling, and so should the projects that reference the static library project.

Use `/translateInclude`

The /translateInclude compiler option (available in the project Property Pages dialog under **C/C++** > **General** > **Translate Includes to Imports**) makes it easier for you to use a header unit library in older projects that `#include` the STL libraries. It makes it unnecessary to change `#include` directives to `import` in your project, while still giving you the advantage of importing the header units instead of including them.

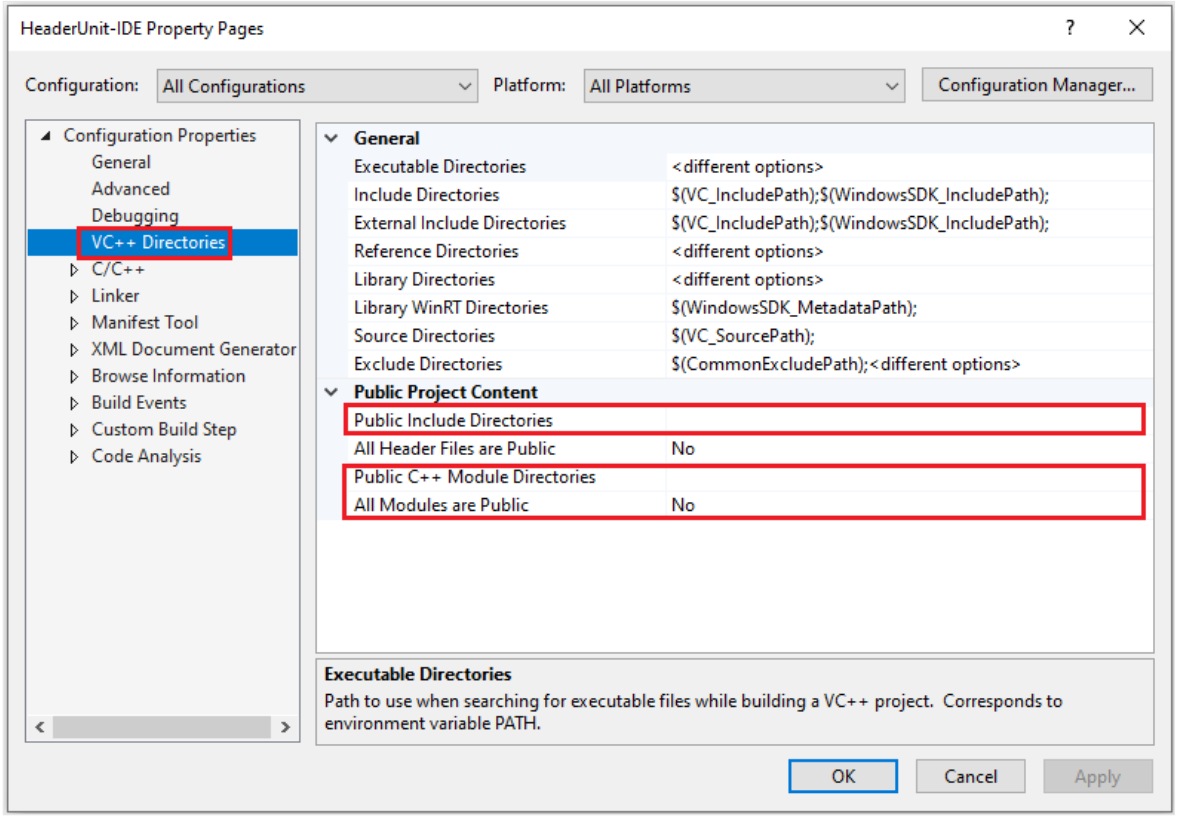
For example, if you have `#include <vector>` in your project and you reference a static library that contains a header unit for `<vector>`, you don't need to manually change `#include <vector>` to `import <vector>;` in your source code. Instead, the compiler automatically treats `#include <vector>` as `import <vector>;`. For more information in detail on this approach, see Approach 2: Scan includes for STL headers to import. Not all STL header files can be compiled to a header unit. The `header-units.json` shipped with Visual Studio lists which STL header files can be compiled into header units. A header that relies on macros to specify its behavior often can't be compiled into a header unit.

An `#include` statement that doesn't refer to a header unit is treated as a normal `#include`.

Reuse header units among projects

Header units built by a static library project are automatically available to all directly and indirectly referencing projects. There are project settings that allow you to select which header units should be automatically available to all referencing projects. The settings are in project settings under **VC++ Directories**.

1. In **Solution Explorer**, right-click the project and select **Properties** to open the project Property Pages dialog.
2. In the left pane of the dialog, select **Configuration Properties** > **VC++ Directories**:



The following properties control the visibility of header units to the build system:

- **Public Include Directories** specifies project directories for header units that should be automatically added to the include path in referencing projects.
- **Public C++ Module Directories** specifies which project directories contain header units that should be available to referencing projects. This property allows you to make some header units public. It's visible to other projects, so put header units that you want to share here. If you use this setting, for convenience, specify **Public Include Directories** to automatically add your public headers to the Include path in referencing projects.

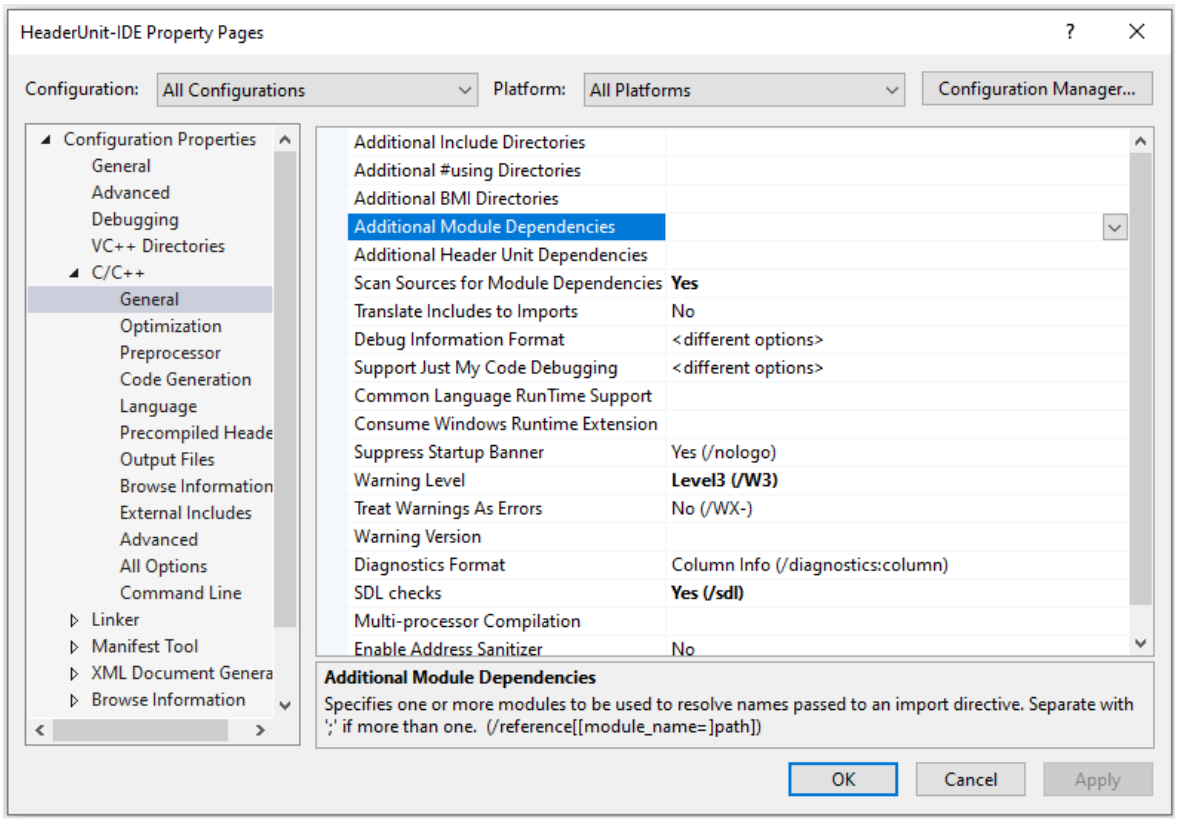
- **All Modules are Public**: when you use header units built as a part of a DLL project, the symbols have to be exported from the DLL. To export module symbols automatically, set this property to **Yes**.

Use a prebuilt module file

Typically, the easiest way to reuse header units among solutions is to reference a shared header unit project from each solution.

If you must use a built header unit that you don't have the project for, you can specify where the built `.ifc` file is so you can import it in your solution. To access this setting:

1. On the main menu, select **Project** > **Properties** to open the project Property Pages dialog.
2. In the left pane of the dialog, select **Configuration Properties** > **C/C++** > **General**.
3. In **Additional Module Dependencies**, add the modules to reference, separated by semicolons. Here's an example of the format to use for **Additional Module Dependencies**: `ModuleName1=Path\To\ModuleName1.ifc;`
 `ModuleName2=Path\To\ModuleName2.ifc`



Select among multiple copies of a header unit

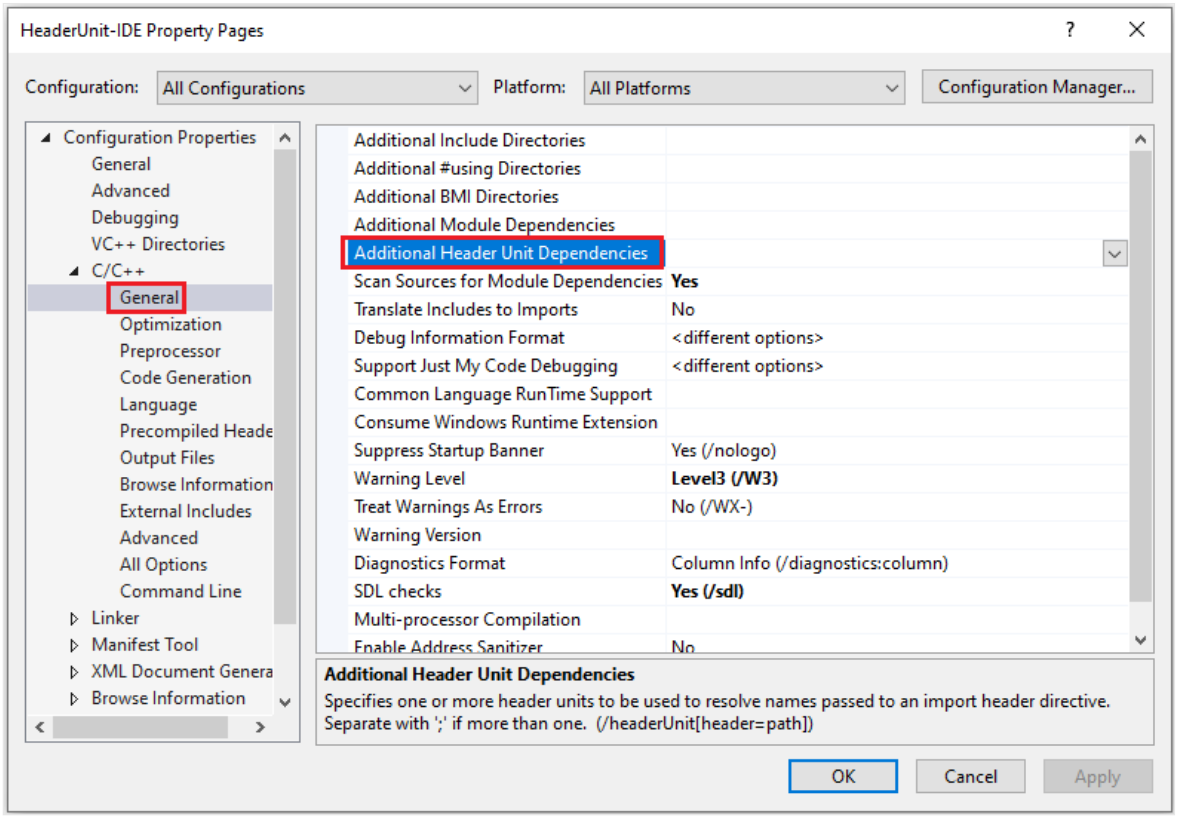
If you reference projects that build multiple header units, either with the same name or for the same header file, you must specify which one to use. You might have different

versions of the header unit built with different compiler settings, for example, and must specify the one that matches your project settings.

Use the project's **Additional Header Unit Dependencies** property to resolve collisions by specifying which header unit to use. Otherwise, it isn't possible to predict which one is picked.

To set the **Additional Header Unit Dependencies** property:

1. On the main menu, select **Project** > **Properties** to open the project Property Pages dialog.
2. In the left pane of the dialog, select **Configuration Properties** > **C/C++** > **General**.
3. Specify which modules or header unit files to use in **Additional Header Unit Dependencies** to resolve collisions. Use this format for **Additional Header Unit Dependencies**: `Path\To\Header1.h= Path\To\HeaderUnit1.ifc;Path\To\Header2.h= Path\To\ HeaderUnit2.ifc`



> **ⓘ Important**
>
> Ensure that projects that share header units are built with compatible compilation options. If you use compilation options when you implement the header unit that are different from the ones you used when you created it, the compiler will issue warnings.

> **Note**
>
> To use header units built as a part of a DLL project, set **All Modules are Public** to **Yes**.

Approach 2: Scan includes for STL headers to import

Another way to import STL libraries is to have Visual Studio scan for the STL headers you `#include` in your project and compile them into header units. The compiler then imports rather than includes those headers.

This option is convenient when your project includes many STL header files across many files, or when build throughput isn't critical. This option doesn't guarantee that a header unit for a particular header file is built only once. However, it's useful if you have a large codebase: You don't have to change your source code to take advantage of the benefits of header units for many of the STL libraries you use.

This approach is less flexible than the static library approach, because it doesn't lend itself towards reusing the built header units in other projects. This approach might not be appropriate for larger projects: It doesn't guarantee an optimal build time, since all of the sources must be scanned for `#include` statements.

Not all header files can be automatically converted to header units. For example, headers that depend on conditional compilation via macros shouldn't be converted to header units. There's an allowlist in the form of a `header-units.json`file for the STL headers that the compiler uses when `/translateInclude` is specified. It determines which STL headers can be compiled into header units. The `header-units.json` file is under the installation directory for Visual Studio. For example, `%ProgramFiles%\Microsoft Visual Studio\2022\Enterprise\VC\Tools\MSVC\14.30.30705\include\header-units.json`. If the STL header file isn't on the list, it's treated as a normal `#include` instead of importing it as a header unit. Another advantage of the `header-units.json` file is that it prevents symbol duplication in the built header units. That is, if compiling a header unit brings in another library header multiple times, the symbols won't be duplicated.

To try out this approach, create a project that includes two STL libraries. Then, change the project's properties so that it imports the libraries as header units instead of including them, as described in the next section.

Create a C++ console app project

Follow these steps to create a project that includes two STL libraries: `<iostream>` and `<vector>`.

1. In Visual Studio, create a new C++ console app project.

2. Replace the contents of the source file as follows:

```
C++

#include <iostream>;
#include <vector>;

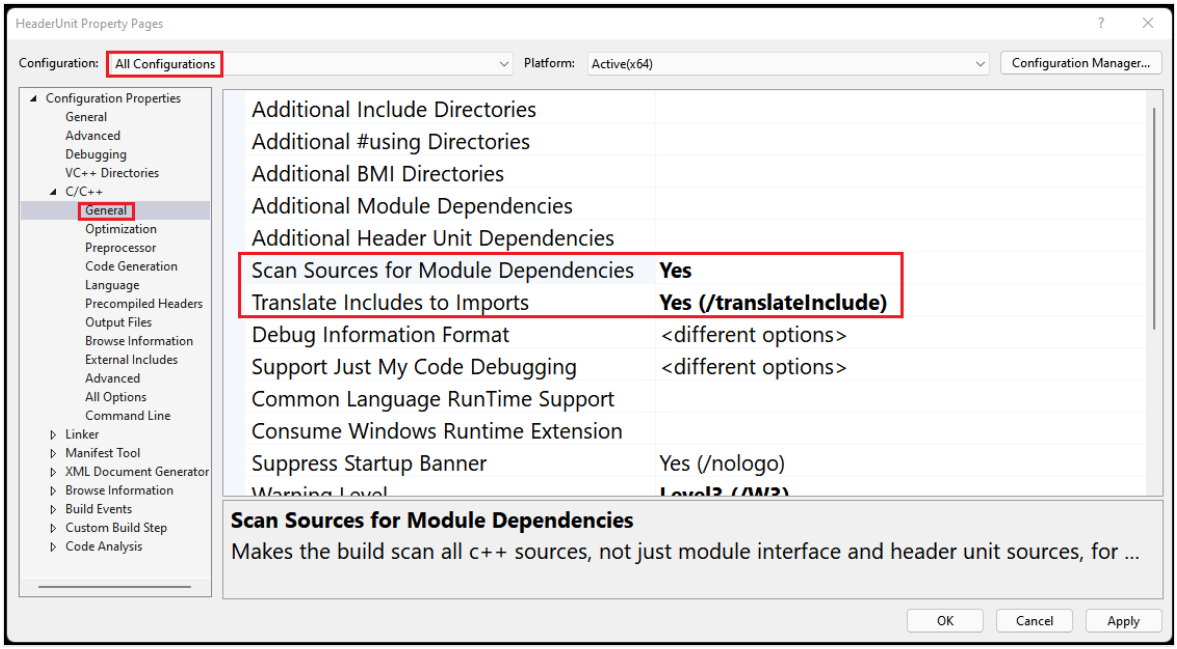
int main()
{
    std::vector<int> numbers = {0, 1, 2};
    std::cout << numbers[1];
}
```

Set project options and run the project

The following steps set the option that causes the compiler to scan for included headers to translate into header units. They also set the option that causes the compiler to treat `#include` as if you had written `import` for header files that can be treated as header units.

1. On the main menu, select **Project** > **Properties** to open the project Property Pages dialog.
2. Select **All Configurations** in the **Configuration** dropdown list, and then select **All Platforms** in the **Platform** dropdown list. These settings ensure that your changes apply whether you're building for debug or release, and other configurations.
3. In the left pane of the dialog, select **Configuration Properties** > **C/C++** > **General**.
4. Set **Scan Sources for Module Dependencies** to **Yes**. This setting ensures that all compatible header files compile into header units.
5. Set **Translate Includes to Imports** to **Yes**. This setting compiles the STL header files listed in the `header-unit.json` file as header units, and then imports them instead

of using the preprocessor to `#include` them.



6. Choose **OK** to save your changes and close the project Property Pages dialog.

The /std:c++20 option or later is required to use header units. To change the C++ language standard used by the compiler:

1. On the main menu, select **Project** > **Properties** to open the project Property Pages dialog.
2. Select **All Configurations** in the **Configuration** dropdown list, and then select **All Platforms** in the **Platform** dropdown list. These settings ensure that your changes apply whether you're building for debug or release, and other configurations.
3. In the left pane of the project Property Pages dialog, select **Configuration Properties** > **General**.
4. In the **C++ Language Standard** dropdown list, select **ISO C++20 Standard (/std:c++20)** (or later).
5. Choose **OK** to save your changes and close the project Property Pages dialog.
6. From the main menu, build the solution by selecting **Build** > **Build Solution**.

Run the solution to verify that it produces the expected output: `1`

The main consideration for whether to use this approach is the balance between convenience and the cost of scanning all your files to determine which header files to build as header units.

See also

Compare header units, modules, and precompiled headers
Tutorial: Import the C++ standard library using modules

Walkthrough: Build and import header units in your Visual C++ projects

/translateInclude

C++ header-units.json reference

Article • 11/09/2022

The `header-units.json` file serves two purposes:

- Specify which header files can be translated into header units when /translateInclude is specified.
- Minimize duplicated symbols to increase build throughput.

This file must be in the same directory as the included header file. This file is only used when /translateInclude is specified along with either /scanDependencies or /sourceDependencies:directives.

Rationale

Some header files can't be safely translated to header units. Header files that depend on macros that aren't defined on the command line, or that aren't defined in the header files included by the header, can't be translated to header units.

If a header defines macros that affect whether other headers are included, it can't be safely translated. For example, given `a.h`, `b.h` and `macros.h`, which are all in the same directory:

C++

```
// a.h

#include "macros.h" // #defines MACRO=1
#ifdef MACRO
#include "b.h"
#endif
```

The `header-units.json` in this directory can contain `a.h` and `b.h`, but not `macros.h`. The `header-units.json` for this example would be similar to this:

JSON

```
{
    "Version": "1.0",
    "BuildAsHeaderUnits": [
        // macros.h should not be listed
        "a.h",
        "b.h"
```

```
    ]
}
```

The reason `macros.h` can't be listed in this `header-units.json` file is that during the scan phase, the header unit (`.ifc`) might not be compiled yet for `macros.h`. In that case, `MACRO` won't be defined when `a.h` is compiled. That means `b.h` will be missing from the list of dependencies for `a.h`. Since it isn't in the list of dependencies, the build system won't build a header unit for `b.h` despite it being listed in the `header-units.json` file.

To avoid this problem, when there's a dependency on a macro in another header file, the header file that defines the macro is excluded from the list of files that can be compiled into a header unit. This way the header file that defines the macro is treated as a normal `#include` and `MACRO` will be visible so that `b.h` is included and listed as one of the dependencies.

Preventing duplicated symbols

The `header-units.json` file is also important because it allows for automatic header unit creation without duplicated symbols. It does this by creating "atomic" header units for the files listed in `header-units.json`. The imported header units don't contain duplicated symbols from the various `#include` directives that were processed while translating the header file.

For example, consider two header files that both include a common header file. Both header files are included by the same source file:

C++

```
// a.h
#include "b.h"

// c.h
#include "b.h"

// Source.cpp
import "a.h";
import "c.h";
```

If the compiler built header units for `a.h`, `b.h` and `c.h`, then the compiled header units `a.h.ifc`, `b.h.ifc`, and `c.h.ifc` would each contain all of the types from `b.h`. Compiling `Source.cpp`, which imports both `a.h` and `c.h`, would require the compiler to deduplicate the `b.h` types, which would impact build performance.

But if there's a `header-units.json` in the `b.h` directory, and `/translateInclude` is specified, then the following happens:

1. The scan of `a.h` and `c.h` lists `b.h` as a header unit import in the dependency scan files generated by the compiler.
2. The build system reads the dependency scan files and determines to build `b.h.ifc` first.
3. Then the build system adds `/headerUnit` for `b.h.ifc` to the command lines for compiling `a.h` and `c.h`. It calls the compiler to build the header units `a.h.ifc` and `c.h.ifc`. Because `/translateInclude` is specified, and `/headerUnit for b.h.ifc` is also specified, `a.h.ifc` and `c.h.ifc` won't contain `b.h` types, so there won't be any duplication in the produced header units.

Schema

There's a `headerunits.json` file for the Standard Template Library (STL) headers. The build system uses it to determine whether to create a header unit for an STL header file, and for its dependencies. If the STL header file isn't on the list, it's treated as a normal `#include` instead of importing it as a header unit.

You can see the `header-units.json` file under the installation directory for Visual Studio. For example: `%ProgramFiles%\Microsoft Visual Studio\2022\Enterprise\VC\Tools\MSVC\14.30.30705\include\header-units.json`

The `header-units.json` file starts with the schema version, followed by an array of filenames for headers that can be built into header units.

The schema also supports comments, as shown here:

JSON

```
{
    "Version": "1.0",
    "BuildAsHeaderUnits": [
        // "__msvc_all_public_headers.hpp", // for testing, not production
        "__msvc_system_error_abi.hpp",
        "__msvc_tzdb.hpp",
        "__msvc_xlocinfo_types.hpp",
        "algorithm",
        "any",
        "array",
        "atomic",
        "barrier",
        "bit",
        "bitset",
```

```
        // "cassert", // design is permanently incompatible with header
units
        ...
}
```

Search rules

The compiler looks for this file in the same directory as the header file being processed. If your library is organized into subdirectories, each subdirectory needs its own `header-units.json` file.

See also

Walkthrough: Import STL libraries as header units
Walkthrough: Build and import header units in your Visual C++ projects

Precompiled header files

Article • 10/16/2023

When you create a new project in Visual Studio, a *precompiled header file* named `pch.h` is added to the project. (In Visual Studio 2017 and earlier, the file was called `stdafx.h`.) The purpose of the file is to speed up the build process. Any stable header files, for example Standard Library headers such as `<vector>`, should be included here. The precompiled header is compiled only when it, or any files it includes, are modified. If you only make changes in your project source code, the build will skip compilation for the precompiled header.

The compiler options for precompiled headers are /Y. In the project property pages, the options are located under **Configuration Properties** > **C/C++** > **Precompiled Headers**. You can choose to not use precompiled headers, and you can specify the header file name and the name and path of the output file.

Custom precompiled code

For large projects that take significant time to build, you may want to consider creating custom precompiled files. The Microsoft C and C++ compilers provide options for precompiling any C or C++ code, including inline code. Using this performance feature, you can compile a stable body of code, store the compiled state of the code in a file, and, during subsequent compilations, combine the precompiled code with code that's still under development. Each later compilation is faster because the stable code doesn't need to be recompiled.

When to precompile source code

Precompiled code is useful during the development cycle to reduce compilation time, especially if:

- You always use a large body of code that changes infrequently.
- Your program comprises multiple modules, all of which use a standard set of include files and the same compilation options. In this case, all include files can be precompiled into one precompiled header. For more information about newer ways to handle include files, see Compare header units, modules, and precompiled headers.

The first compilation (the one that creates the precompiled header file) takes a bit longer than subsequent compilations. Subsequent compilations can proceed more quickly by including the precompiled code.

You can precompile both C and C++ programs. In C++ programming, it's common practice to separate class interface information into header files. These header files can later be included in programs that use the class. By precompiling these headers, you can reduce the time a program takes to compile.

> **Note**
>
> Although you can use only one precompiled header (`.pch`) file per source file, you can use multiple `.pch` files in a project.

Two choices for precompiling code

You can precompile any C or C++ code; you're not limited to precompiling only header files.

Precompiling requires planning, but it offers much faster compilations if you precompile source code other than simple header files.

Precompile code when you know that your source files use common sets of header files, or when you want to include source code in your precompilation.

The precompiled-header options are /Yc (Create Precompiled Header File) and /Yu (Use Precompiled Header File). Use `/Yc` to create a precompiled header. When used with the optional hdrstop pragma, `/Yc` lets you precompile both header files and source code. Select `/Yu` to use an existing precompiled header in the existing compilation. You can also use `/Fp` with the `/Yc` and `/Yu` options to provide an alternative name for the precompiled header.

The compiler option reference articles for `/Yu` and `/Yc` discuss how to access this functionality in the development environment.

Precompiled header consistency rules

Because PCH files contain information about the machine environment and memory address information about the program, you should only use a PCH file on the machine where it was created.

Consistency rules for per-file use of precompiled headers

The /Yu compiler option lets you specify which PCH file to use.

When you use a PCH file, the compiler assumes the same compilation environment that was in effect when you created the PCH file, unless you specify otherwise. The compilation environment includes the compiler options, pragmas, and so on. If the compiler detects an inconsistency, it issues a warning and identifies the inconsistency where possible. Such warnings don't necessarily indicate a problem with the PCH file; they simply warn you of possible conflicts. Consistency requirements for PCH files are described in the following sections.

Compiler option consistency

The following compiler options can trigger an inconsistency warning when using a PCH file:

- Macros created using the Preprocessor (`/D`) option must be the same between the compilation that created the PCH file and the current compilation. The state of defined constants isn't checked, but unpredictable results can occur if these macros change.
- PCH files don't work with the `/E` and `/EP` options.
- PCH files must be created using either the Generate Browse Info (`/FR`) option or the Exclude Local Variables (`/Fr`) option before subsequent compilations that use the PCH file can use these options.

C 7.0-compatible (`/Z7`)

If this option is in effect when the PCH file is created, later compilations that use the PCH file can use the debugging information.

If the C 7.0-Compatible (`/Z7`) option isn't in effect when the PCH file is created, later compilations that use the PCH file and `/Z7` trigger a warning. The debugging information is placed in the current *`.obj`* file, and local symbols defined in the PCH file aren't available to the debugger.

Include path consistency

A PCH file doesn't contain information about the header include path that was in effect when it was created. When you use a PCH file, the compiler always uses the header include path specified in the current compilation.

Source file consistency

When you specify the Use Precompiled Header File (`/Yu`) option, the compiler ignores all preprocessor directives (including pragmas) that appear in the source code that will be precompiled. The compilation specified by such preprocessor directives must be the same as the compilation used for the Create Precompiled Header File (`/Yc`) option.

Pragma consistency

Pragmas processed during the creation of a PCH file usually affect the file with which the PCH file is later used. The `comment` and `message` pragmas don't affect the remainder of the compilation.

These pragmas affect only the code within the PCH file; they don't affect code that later uses the PCH file:

`comment`
`linesize`

`message`
`page`

`pagesize`
`skip`

`subtitle`
`title`

These pragmas are retained as part of a precompiled header, and affect the remainder of a compilation that uses the precompiled header:

`alloc_text`
`auto_inline`
`check_stack`
`code_seg`
`data_seg`

`function`

`include_alias`

`init_seg`

`inline_depth`

`inline_recursion`

`intrinsic`

`optimize`

`pack`

`pointers_to_members`

`setlocale`

`vtordisp`

`warning`

Consistency rules for /Yc and /Yu

When you use a precompiled header created using `/Yc` or `/Yu`, the compiler compares the current compilation environment to the one that existed when you created the PCH file. Be sure to specify an environment consistent with the previous one (using consistent compiler options, pragmas, and so on) for the current compilation. If the compiler detects an inconsistency, it issues a warning and identifies the inconsistency where possible. Such warnings don't necessarily indicate a problem with the PCH file; they simply warn you of possible conflicts. The following sections explain the consistency requirements for precompiled headers.

Compiler option consistency

This table lists compiler options that might trigger an inconsistency warning when using a precompiled header:

Option	Name	Rule
`/D`	Define constants and macros	Must be the same between the compilation that created the precompiled header and the current compilation. The state of defined constants isn't checked. However, unpredictable results can occur if your files depend on the values of the changed constants.
`/E` or `/EP`	Copy preprocessor	Precompiled headers don't work with the `/E` or `/EP` option.

Option	Name	Rule
	output to standard output	
`/Fr` or `/FR`	Generate Microsoft Source Browser information	For the `/Fr` and `/FR` options to be valid with the `/Yu` option, they must also have been in effect when the precompiled header was created. Subsequent compilations that use the precompiled header also generate Source Browser information. Browser information is placed in a single *`.sbr`* file and is referenced by other files in the same manner as CodeView information. You can't override the placement of Source Browser information.
`/GA`, `/GD`, `/GE`, `/Gw`, or `/GW`	Windows protocol options	Must be the same between the compilation that created the precompiled header and the current compilation. The compiler emits a warning if these options differ.
`/Zi`	Generate complete debugging information	If this option is in effect when the precompiled header is created, subsequent compilations that use the precompilation can use that debugging information. If `/Zi` isn't in effect when the precompiled header is created, subsequent compilations that use the precompilation and the `/Zi` option trigger a warning. The debugging information is placed in the current object file, and local symbols defined in the precompiled header aren't available to the debugger.

> **Note**
>
> The precompiled header facility is intended for use only in C and C++ source files.

Using precompiled headers in a project

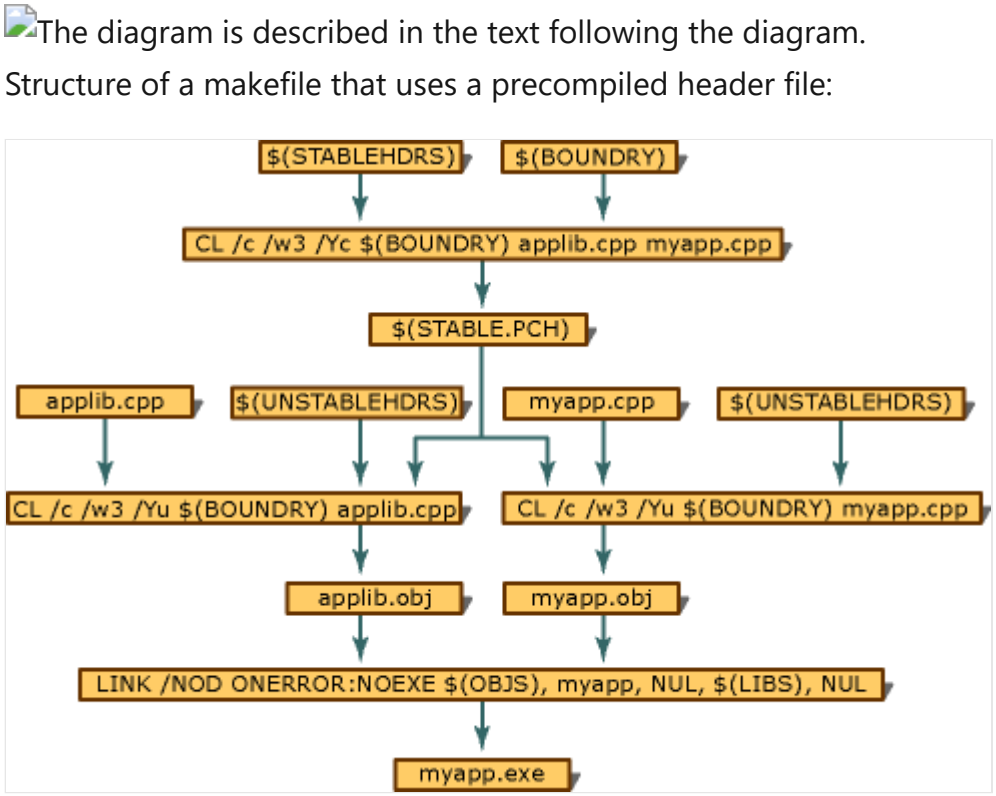
Previous sections present an overview of precompiled headers: /Yc and /Yu, the /Fp option, and the hdrstop pragma. This section describes a method for using the manual precompiled-header options in a project; it ends with an example makefile and the code that it manages.

For another approach to using the manual precompiled-header options in a project, study one of the makefiles located in the *`MFC\SRC`* directory that's created during the default setup of Visual Studio. These makefiles take a similar approach to the one presented in this section. They make greater use of Microsoft Program Maintenance Utility (NMAKE) macros, and offer greater control of the build process.

PCH files in the build process

The code base of a software project is often contained in multiple C or C++ source files, object files, libraries, and header files. Typically, a makefile coordinates the combination of these elements into an executable file. The following figure shows the structure of a makefile that uses a precompiled header file. The NMAKE macro names and the file names in this diagram are consistent with the example code found in Sample makefile for PCH and Example code for PCH.

The figure uses three diagrammatic devices to show the flow of the build process. Named rectangles represent each file or macro; the three macros represent one or more files. Shaded areas represent each compile or link action. Arrows show which files and macros are combined during the compilation or linking process.



Beginning at the top of the diagram, both `STABLEHDRS` and `BOUNDRY` are NMAKE macros in which you list files not likely to need recompilation. These files are compiled by the command string

```
CL /c /W3 /Yc$(BOUNDRY) applib.cpp myapp.cpp
```

only if the precompiled header file (*`STABLE.pch`*) doesn't exist or if you make changes to the files listed in the two macros. In either case, the precompiled header file will contain code only from the files listed in the `STABLEHDRS` macro. List the last file you want precompiled in the `BOUNDRY` macro.

The files you list in these macros can be either header files or C or C++ source files. (A single PCH file can't be used with both C and C++ sources.) You can use the **`hdrstop`**

macro to stop precompilation at some point within the `BOUNDRY` file. For more information, see hdrstop.

Next in the diagram, *`APPLIB.obj`* represents the support code used in your final application. It's created from *`APPLIB.cpp`*, the files listed in the `UNSTABLEHDRS` macro, and precompiled code from the precompiled header.

`MYAPP.obj` represents your final application. It's created from *`MYAPP.cpp`*, the files listed in the `UNSTABLEHDRS` macro, and precompiled code from the precompiled header.

Finally, the executable file (*`MYAPP.EXE`*) is created by linking the files listed in the `OBJS` macro (*`APPLIB.obj`* and *`MYAPP.obj`*).

Sample makefile for PCH

The following makefile uses macros and an `!IF`, `!ELSE`, `!ENDIF` flow-of-control command structure to simplify its adaptation to your project.

NMAKE

```
# Makefile : Illustrates the effective use of precompiled
#            headers in a project
# Usage:     NMAKE option
# option:    DEBUG=[0|1]
#            (DEBUG not defined is equivalent to DEBUG=0)
#
OBJS = myapp.obj applib.obj
# List all stable header files in the STABLEHDRS macro.
STABLEHDRS = stable.h another.h
# List the final header file to be precompiled here:
BOUNDRY = stable.h
# List header files under development here:
UNSTABLEHDRS = unstable.h
# List all compiler options common to both debug and final
# versions of your code here:
CLFLAGS = /c /W3
# List all linker options common to both debug and final
# versions of your code here:
LINKFLAGS = /nologo
!IF "$(DEBUG)" == "1"
CLFLAGS   = /D_DEBUG $(CLFLAGS) /Od /Zi
LINKFLAGS = $(LINKFLAGS) /COD
LIBS      = slibce
!ELSE
CLFLAGS   = $(CLFLAGS) /Oselg /Gs
LINKFLAGS = $(LINKFLAGS)
LIBS      = slibce
!ENDIF
myapp.exe: $(OBJS)
```

```
    link $(LINKFLAGS) @<<
$(OBJS), myapp, NUL, $(LIBS), NUL;
<<
# Compile myapp
myapp.obj  : myapp.cpp $(UNSTABLEHDRS)  stable.pch
    $(CPP) $(CLFLAGS) /Yu$(BOUNDRY)    myapp.cpp
# Compile applib
applib.obj : applib.cpp $(UNSTABLEHDRS) stable.pch
    $(CPP) $(CLFLAGS) /Yu$(BOUNDRY)    applib.cpp
# Compile headers
stable.pch : $(STABLEHDRS)
    $(CPP) $(CLFLAGS) /Yc$(BOUNDRY)    applib.cpp myapp.cpp
```

Aside from the `STABLEHDRS`, `BOUNDRY`, and `UNSTABLEHDRS` macros shown in the figure "Structure of a Makefile That Uses a Precompiled Header File" in PCH files in the build process, this makefile provides a `CLFLAGS` macro and a `LINKFLAGS` macro. You must use these macros to list compiler and linker options that apply whether you build a debug or final version of the application's executable file. There's also a `LIBS` macro where you list the libraries your project requires.

The makefile also uses `!IF`, `!ELSE`, `!ENDIF` to detect whether you define a `DEBUG` symbol on the NMAKE command line:

NMAKE

```
NMAKE DEBUG=[1|0]
```

This feature makes it possible for you to use the same makefile during development and for the final versions of your program. Use `DEBUG=0` for the final versions. The following command lines are equivalent:

NMAKE

```
NMAKE
NMAKE DEBUG=0
```

For more information on makefiles, see NMAKE reference. Also see MSVC compiler options and the MSVC linker options.

Example code for PCH

The following source files are used in the makefile described in PCH files in the build process and Sample makefile for PCH. The comments contain important information.

Source file `ANOTHER.H`:

```
C++
```

```
// ANOTHER.H : Contains the interface to code that is not
//             likely to change.
//
#ifndef __ANOTHER_H
#define __ANOTHER_H
#include<iostream>
void savemoretime( void );
#endif // __ANOTHER_H
```

Source file `STABLE.H`:

```
C++
```

```
// STABLE.H : Contains the interface to code that is not likely
//            to change. List code that is likely to change
//            in the makefile's STABLEHDRS macro.
//
#ifndef __STABLE_H
#define __STABLE_H
#include<iostream>
void savetime( void );
#endif // __STABLE_H
```

Source file `UNSTABLE.H`:

```
C++
```

```
// UNSTABLE.H : Contains the interface to code that is
//              likely to change. As the code in a header
//              file becomes stable, remove the header file
//              from the makefile's UNSTABLEHDR macro and list
//              it in the STABLEHDRS macro.
//
#ifndef __UNSTABLE_H
#define __UNSTABLE_H
#include<iostream>
void notstable( void );
#endif // __UNSTABLE_H
```

Source file `APPLIB.CPP`:

```
C++
```

```
// APPLIB.CPP : This file contains the code that implements
//              the interface code declared in the header
```

```
//              files STABLE.H, ANOTHER.H, and UNSTABLE.H.
//
#include"another.h"
#include"stable.h"
#include"unstable.h"
using namespace std;
// The following code represents code that is deemed stable and
// not likely to change. The associated interface code is
// precompiled. In this example, the header files STABLE.H and
// ANOTHER.H are precompiled.
void savetime( void )
    { cout << "Why recompile stable code?\n"; }
void savemoretime( void )
    { cout << "Why, indeed?\n\n"; }
// The following code represents code that is still under
// development. The associated header file is not precompiled.
void notstable( void )
    { cout << "Unstable code requires"
           << " frequent recompilation.\n";
    }
```

Source file `MYAPP.CPP`:

C++

```
// MYAPP.CPP : Sample application
//             All precompiled code other than the file listed
//             in the makefile's BOUNDRY macro (stable.h in
//             this example) must be included before the file
//             listed in the BOUNDRY macro. Unstable code must
//             be included after the precompiled code.
//
#include"another.h"
#include"stable.h"
#include"unstable.h"
int main( void )
{
    savetime();
    savemoretime();
    notstable();
}
```

See also

Compare header units, modules, and precompiled headers
C/C++ building reference
MSVC compiler options Overview of modules in C++
Tutorial: Import the C++ standard library using modules

Walkthrough: Build and import header units in your Visual C++ projects

Walkthrough: Import STL libraries as header units

Release Builds

Article • 08/03/2021

A release build uses optimizations. When you use optimizations to create a release build, the compiler will not produce symbolic debugging information. The absence of symbolic debugging information, along with the fact that code is not generated for TRACE and ASSERT calls, means that the size of your executable file is reduced and will therefore be faster.

In this section

Common Problems When Creating a Release Build
Fixing Release Build Problems
Using VERIFY Instead of ASSERT
Using the Debug Build to Check for Memory Overwrite
How to: Debug a Release Build
Checking for Memory Overwrites
Optimizing Your Code

See also

C/C++ Building Reference

How to: Create a Release Build

Article • 08/03/2021

To generate a release build of your program

1. Select **Release** from the **Solution Configuration** drop-down list, which is on the **Standard** toolbar.

2. On the **Build** menu, click **Build**.

See also

Release Builds

Common Problems When Creating a Release Build

Article • 08/03/2021

During development, you will usually build and test with a debug build of your project. If you then build your application for a release build, you may get an access violation.

The list below shows the primary differences between a debug and a release (nondebug) build. There are other differences, but following are the primary differences that would cause an application to fail in a release build when it works in a debug build.

- Heap Layout
- Compilation
- Pointer Support
- Optimizations

See the /GZ (Catch Release-Build Errors in Debug Build) compiler option for information on how to catch release build errors in debug builds.

Heap Layout

Heap layout will be the cause of about ninety percent of the apparent problems when an application works in debug, but not release.

When you build your project for debug, you are using the debug memory allocator. This means that all memory allocations have guard bytes placed around them. These guard bytes detect a memory overwrite. Because heap layout is different between release and debug versions, a memory overwrite might not create any problems in a debug build, but may have catastrophic effects in a release build.

For more information, see Check for Memory Overwrite and Use the Debug Build To Check for Memory Overwrite.

Compilation

Many of the MFC macros and much of the MFC implementation changes when you build for release. In particular, the ASSERT macro evaluates to nothing in a release build,

so none of the code found in ASSERTs will be executed. For more information, see Examine ASSERT Statements.

Some functions are inlined for increased speed in the release build. Optimizations are generally turned on in a release build. A different memory allocator is also being used.

Pointer Support

The lack of debugging information removes the padding from your application. In a release build, stray pointers have a greater chance of pointing to uninitialized memory instead of pointing to debug information.

Optimizations

Depending on the nature of certain segments of code, the optimizing compiler might generate unexpected code. This is the least likely cause of release build problems, but it does arise on occasion. For a solution, see Optimizing Your Code.

See also

Release Builds
Fixing Release Build Problems

Fixing Release Build Problems

Article • 08/03/2021

If your code generates compile errors after switching from debug build to release build, there are some areas you should check.

You may receive compiler warnings during an optimized (release) build that you did not receive during a debug build.

- Examine ASSERT Statements
- Use the Debug Build To Check for Memory Overwrites
- Turn on Generation of Debug Information for the Release Build
- Check for Memory Overwrite

See also

Release Builds
Common Problems When Creating a Release Build
Optimizing Your Code

Using VERIFY Instead of ASSERT

Article • 08/03/2021

Suppose that when you run the debug version of your MFC application, there are no problems. However, the release version of the same application crashes, returns incorrect results, and/or exhibits some other abnormal behavior.

This problem can be caused when you place important code in an ASSERT statement to verify that it performs correctly. Because ASSERT statements are commented out in a release build of an MFC program, the code does not run in a release build.

If you are using ASSERT to confirm that a function call succeeded, consider using VERIFY instead. The VERIFY macro evaluates its own arguments in both debug and release builds of the application.

Another preferred technique is to assign the function's return value to a temporary variable and then test the variable in an ASSERT statement.

Examine the following code fragment:

```
enum {
    sizeOfBuffer = 20
};
char *buf;
ASSERT(buf = (char *) calloc(sizeOfBuffer, sizeof(char) ));
strcpy_s( buf, sizeOfBuffer, "Hello, World" );
free( buf );
```

This code runs perfectly in a debug version of an MFC application. If the call to `calloc( )` fails, a diagnostic message that includes the file and line number appears. However, in a retail build of an MFC application:

- the call to `calloc( )` never occurs, leaving `buf` uninitialized, or
- `strcpy_s( )` copies "`Hello, World`" into a random piece of memory, possibly crashing the application or causing the system to stop responding, or
- `free()` attempts to free memory that was never allocated.

To use ASSERT correctly, the code sample should be changed to the following:

```
enum {
    sizeOfBuffer = 20
};
char *buf;
buf = (char *) calloc(sizeOfBuffer, sizeof(char) );
ASSERT( buf != NULL );
strcpy_s( buf, sizeOfBuffer, "Hello, World" );
free( buf );
```

Or, you can use VERIFY instead:

```
enum {
    sizeOfBuffer = 20
};
char *buf;
VERIFY(buf = (char *) calloc(sizeOfBuffer, sizeof(char) ));
strcpy_s( buf, sizeOfBuffer, "Hello, World" );
free( buf );
```

See also

Fixing Release Build Problems

Using the Debug Build to Check for Memory Overwrite

Article • 08/03/2021

To use the debug build to check for memory overwrite, you must first rebuild your project for debug. Then, go to the very beginning of your application's `InitInstance` function and add the following line:

```
afxMemDF |= checkAlwaysMemDF;
```

The debug memory allocator puts guard bytes around all memory allocations. However, these guard bytes don't do any good unless you check whether they have been changed (which would indicate a memory overwrite). Otherwise, this just provides a buffer that might, in fact, allow you to get away with a memory overwrite.

By turning on the `checkAlwaysMemDF`, you will force MFC to make a call to the `AfxCheckMemory` function every time a call to **`new`** or **`delete`** is made. If a memory overwrite was detected, it will generate a TRACE message that looks similar to the following:

```
Damage Occurred! Block=0x5533
```

If you see one of these messages, you need to step through your code to determine where the damage occurred. To isolate more precisely where the memory overwrite occurred, you can make explicit calls to `AfxCheckMemory` yourself. For example:

```
ASSERT(AfxCheckMemory());
    DoABunchOfStuff();
    ASSERT(AfxCheckMemory());
```

If the first ASSERT succeeds and the second one fails, it means that the memory overwrite must have occurred in the function between the two calls.

Depending on the nature of your application, you may find that `afxMemDF` causes your program to run too slowly to even test. The `afxMemDF` variable causes `AfxCheckMemory` to

be called for every call to new and delete. In that case, you should scatter your own calls to `AfxCheckMemory()` as shown above, and try to isolate the memory overwrite that way.

See also

Fixing Release Build Problems

How to: Debug a Release Build

Article • 08/03/2021

You can debug a release build of an application.

To debug a release build

1. Open the **Property Pages** dialog box for the project. For details, see Set C++ compiler and build properties in Visual Studio.

2. Click the **C/C++** node. Set **Debug Information Format** to C7 compatible (/Z7) or **Program Database (/Zi)**.

3. Expand **Linker** and click the **General** node. Set **Enable Incremental Linking** to No (/INCREMENTAL:NO).

4. Select the **Debugging** node. Set **Generate Debug Info** to Yes (/DEBUG).

5. Select the **Optimization** node. Set **References** to /OPT:REF and **Enable COMDAT Folding** to /OPT:ICF.

6. You can now debug your release build application. To find a problem, step through the code (or use Just-In-Time debugging) until you find where the failure occurs, and then determine the incorrect parameters or code.

 If an application works in a debug build, but fails in a release build, one of the compiler optimizations may be exposing a defect in the source code. To isolate the problem, disable selected optimizations for each source code file until you locate the file and the optimization that is causing the problem. (To expedite the process, you can divide the files into two groups, disable optimization on one group, and when you find a problem in a group, continue dividing until you isolate the problem file.)

 You can use /RTC to try to expose such bugs in your debug builds.

 For more information, see Optimizing Your Code.

See also

Fixing Release Build Problems

Checking for Memory Overwrites

Article • 08/03/2021

If you get an access violation on a call to a heap manipulation function, it is possible that your program has corrupted the heap. A common symptom of this situation would be:

```
Access Violation in _searchseg
```

The _heapchk function is available in both debug and release builds (Windows NT only) for verifying the integrity of the run time library heap. You can use `_heapchk` in much the same way as the `AfxCheckMemory` function to isolate a heap overwrite, for example:

```
if(_heapchk()!=_HEAPOK)
   DebugBreak();
```

If this function ever fails, you need to isolate at which point the heap was corrupted.

See also

Fixing Release Build Problems

Optimizing your code

Article • 08/03/2021

By optimizing an executable, you can achieve a balance between fast execution speed and small code size. This topic discusses some of the mechanisms that Visual Studio provides to help you optimize code.

Language features

The following topics describe some of the optimization features in the C/C++ language.

Optimization Pragmas and Keywords
A list of keywords and pragmas that you can use in your code to improve performance.

Compiler Options Listed by Category
A list of **/O** compiler options that specifically affect execution speed or code size.

Rvalue Reference Declarator: &&
Rvalue references support the implementation of *move semantics*. If move semantics are used to implement template libraries, the performance of applications that use those templates can significantly improve.

The optimize pragma

If an optimized section of code causes errors or a slowdown, you can use the optimize pragma to turn off optimization for that section.

Enclose the code between two pragmas, as shown here:

C++

```
#pragma optimize("", off)
// some code here
#pragma optimize("", on)
```

Programming practices

You might notice additional warning messages when you compile your code with optimization. This behavior is expected because some warnings relate only to optimized code. You can avoid many optimization problems if you heed these warnings.

Paradoxically, optimizing a program for speed could cause code to run slower. This is because some optimizations for speed increase code size. For example, inlining functions eliminates the overhead of function calls. However, inlining too much code might make your program so large that the number of virtual-memory page faults increases. Therefore, the speed gained from eliminating function calls may be lost to memory swapping.

The following topics discuss good programming practices.

Tips for Improving Time-Critical Code
Better coding techniques can yield better performance. This topic suggests coding techniques that can help you make sure that the time-critical parts of your code perform satisfactorily.

Optimization Best Practices
Provides general guidelines about how best to optimize your application.

Debugging optimized code

Because optimization might change the code created by the compiler, we recommend that you debug your application and measure its performance, and then optimize your code.

The following topics provide information about how to debug release builds.

- Debugging in Visual Studio
- How to: Debug Optimized Code
- Why Floating-Point Numbers May Lose Precision

The following topics provide information about how to optimize building, loading, and executing your code.

- Improving Compiler Throughput
- Using Function Name Without () Produces No Code
- Optimizing Inline Assembly
- Specifying Compiler Optimization for an ATL Project
- What optimization techniques should I use to improve the client application's performance when loading?

In this section

Optimization Pragmas and Keywords
Improving Compiler Throughput
Why Floating-Point Numbers May Lose Precision
IEEE Floating-Point Representation
Tips for Improving Time-Critical Code
Using Function Name Without () Produces No Code
Optimization Best Practices
Profile-Guided Optimizations
Environment Variables for Profile-Guided Optimizations
PgoAutoSweep
pgomgr
pgosweep
How to: Merge Multiple PGO Profiles into a Single Profile

See also

C/C++ Building Reference

Optimization Pragmas and Keywords

Article • 08/03/2021

Several keywords and pragmas that you use in your C or C++ code affect optimization:

- __asm
- __assume
- inline, __inline, or __forceinline
- #pragma auto_inline
- #pragma check_stack
- #pragma function
- #pragma inline_depth
- #pragma inline_recursion
- #pragma intrinsic
- #pragma optimize
- register Keyword

See also

Optimizing Your Code

Improving Compiler Throughput

Article • 08/03/2021

Use precompiled header files to build your project faster. This is important if you are using ATL, MFC, or the Windows SDK header files.

See /Yc and /Yu.

For more information on precompiled headers, see Precompiled Header Files.

See also

Optimizing Your Code

Why Floating-Point Numbers May Lose Precision

Article • 08/03/2021

Floating-point decimal values generally do not have an exact binary representation. This is a side effect of how the CPU represents floating point data. For this reason, you may experience some loss of precision, and some floating-point operations may produce unexpected results.

This behavior is the result of one of the following:

- The binary representation of the decimal number may not be exact.
- There is a type mismatch between the numbers used (for example, mixing float and double).

To resolve the behavior, most programmers either ensure that the value is greater or less than what is needed, or they get and use a Binary Coded Decimal (BCD) library that will maintain the precision.

Binary representation of floating-point values affects the precision and accuracy of floating-point calculations. Microsoft Visual C++ uses IEEE floating-point format.

Example

C

```c
// Floating-point_number_precision.c
// Compile options needed: none. Value of c is printed with a decimal
// point precision of 10 and 6 (printf rounded value by default) to
// show the difference
#include <stdio.h>

#define EPSILON 0.0001   // Define your own tolerance
#define FLOAT_EQ(x,v) (((v - EPSILON) < x) && (x <( v + EPSILON)))

int main() {
   float a, b, c;

   a = 1.345f;
   b = 1.123f;
   c = a + b;
   // if (FLOAT_EQ(c, 2.468)) // Remove comment for correct result
   if (c == 2.468)            // Comment this line for correct result
      printf_s("They are equal.\n");
```

```
    else
       printf_s("They are not equal! The value of c is %13.10f "
                "or %f",c,c);
}
```

Output

```
They are not equal! The value of c is  2.4679999352 or 2.468000
```

Comments

For EPSILON, you can use the constants FLT_EPSILON, which is defined for float as 1.192092896e-07F, or DBL_EPSILON, which is defined for double as 2.2204460492503131e-016. You need to include float.h for these constants. These constants are defined as the smallest positive number x, such that x+1.0 is not equal to 1.0. Because this is a very small number, you should employ user-defined tolerance for calculations involving very large numbers.

See also

Optimizing Your Code

IEEE Floating-Point Representation

Article • 08/03/2021

Microsoft C++ (MSVC) is consistent with the IEEE numeric standards. The IEEE-754 standard describes floating-point formats, a way to represent real numbers in hardware. There are at least five internal formats for floating-point numbers that are representable in hardware targeted by the MSVC compiler. The compiler only uses two of them. The *single-precision* (4-byte) and *double-precision* (8-byte) formats are used in MSVC. Single-precision is declared using the keyword `float`. Double-precision is declared using the keyword `double`. The IEEE standard also specifies *half-precision* (2-byte) and *quadruple-precision* (16-byte) formats, and a *double-extended-precision* (10-byte) format, which some C and C++ compilers implement as the `long double` data type. In the MSVC compiler, the `long double` data type is treated as a distinct type, but the storage type maps to `double`. There is, however, intrinsic and assembly language support for computations using the other formats, including the double-extended-precision format, where supported by hardware.

The values are stored as follows:

Value	Stored as
single-precision	sign bit, 8-bit exponent, 23-bit significand
double-precision	sign bit, 11-bit exponent, 52-bit significand

In single-precision and double-precision formats, there's an assumed leading 1 in the fractional part. The fractional part is called the *significand* (sometimes known as the *mantissa*). This leading 1 isn't stored in memory, so the significands are actually 24 or 53 bits, even though one less bit gets stored. The double-extended-precision format actually stores this bit.

The exponents are biased by half of their possible value. It means you subtract this bias from the stored exponent to get the actual exponent. If the stored exponent is less than the bias, it's actually a negative exponent.

The exponents are biased as follows:

Exponent	Biased by
8-bit (single-precision)	127
11-bit (double-precision)	1023

These exponents aren't powers of ten; they're powers of two. That is, 8-bit stored exponents can range from -127 to 127, stored as 0 to 254. The value 2^{127} is roughly equivalent to 10^{38}, which is the actual limit of single-precision.

The significand is stored as a binary fraction of the form 1.XXX... . This fraction has a value greater than or equal to 1 and less than 2. Real numbers are always stored in *normalized form*. That is, the significand is left-shifted such that the high-order bit of the significand is always 1. Because this bit is *always* 1, it's assumed (not stored) in the single-precision and double-precision formats. The binary (not decimal) point is assumed to be just to the right of the leading 1.

The format for floating-point representation is as follows:

Format	**byte 1**	**byte 2**	**byte 3**	**byte 4**	**...**	**byte n**
single-precision	`SXXXXXXX`	`XMMMMMMM`	`MMMMMMMM`	`MMMMMMMM`		
double-precision	`SXXXXXXX`	`XXXXMMMM`	`MMMMMMMM`	`MMMMMMMM`	...	`MMMMMMMM`

`S` represents the sign bit, the `X`'s are the biased exponent bits, and the `M`'s are the significand bits. The leftmost bit is assumed in single-precision and double-precision formats.

To shift the binary point properly, you first unbias the exponent and then move the binary point to the right or left the appropriate number of bits.

Special values

The floating-point formats include some values that are treated specially.

Zero

Zero can't be normalized, which makes it unrepresentable in the normalized form of a single-precision or double-precision value. A special bit pattern of all zeroes represents 0. It's also possible to represent -0 as zero with the sign bit set, but -0 and 0 always compare as equal.

Infinities

The $+\infty$ and $-\infty$ values are represented by an exponent of all ones, and a significand that's all zeroes. Positive and negative are represented by using the sign bit.

Subnormals

It's possible to represent numbers of smaller magnitude than the smallest number in normalized form. They're called *subnormal* or *denormal* numbers. If the exponent is all zeroes and the significand is non-zero, then implicit leading bit of the significand is considered to be zero, not one. The precision of subnormal numbers goes down as the number of leading zeroes in the significand goes up.

NaN - Not a Number

It's possible to represent values that aren't real numbers, such as 0 / 0, in the IEEE floating-point format. A value of this kind is called a *NaN*. A NaN is represented by an exponent of all ones and a non-zero significand. There are two kinds of NaNs, *quiet* NaNs, or QNaNs, and *signaling* NaNs, or SNaNs. Quiet NaNs have a leading one in the significand, and get propagated through an expression. They represent an indeterminate value, such as the result of dividing by infinity, or multiplying an infinity by zero. Signaling NaNs have a leading zero in the significand. They're used for operations that aren't valid, to signal a floating-point hardware exception.

Examples

The following are some examples in single-precision format:

- For the value 2, the sign bit is zero. The stored exponent is 128, or 1000 0000 in binary, which is 127 plus 1. The stored binary significand is (1.) 000 0000 0000 0000 0000 0000, which has an implied leading 1 and binary point, so the actual significand is one.

Value	Formula	Binary representation	Hexadecimal
2	$1 * 2^1$	0100 0000 0000 0000 0000 0000 0000 0000	0x40000000

- The value -2. Same as +2 except that the sign bit is set. The same thing is true for the negative of all IEEE format floating-point numbers.

Value	Formula	Binary representation	Hexadecimal
-2	$-1 * 2^1$	1100 0000 0000 0000 0000 0000 0000 0000	0xC0000000

- The value 4. Same significand, exponent increases by one (biased value is 129, or 100 0000 1 in binary.

Value	Formula	Binary representation	Hexadecimal
4	$1 * 2^2$	0100 0000 1000 0000 0000 0000 0000 0000	0x40800000

- The value 6. Same exponent, significand is larger by half. It's (1.) 100 0000 ... 0000 0000, which, since it's a binary fraction, is 1 1/2 because the values of the fractional digits are 1/2, 1/4, 1/8, and so forth.

Value	Formula	Binary representation	Hexadecimal
6	$1.5 * 2^2$	0100 0000 1100 0000 0000 0000 0000 0000	0x40C00000

- The value 1. Same significand as other powers of two, the biased exponent is one less than two at 127, or 011 1111 1 in binary.

Value	Formula	Binary representation	Hexadecimal
1	$1 * 2^0$	0011 1111 1000 0000 0000 0000 0000 0000	0x3F800000

- The value 0.75. The biased exponent is 126, 011 1111 0 in binary, and the significand is (1.) 100 0000 ... 0000 0000, which is 1 1/2.

Value	Formula	Binary representation	Hexadecimal
0.75	$1.5 * 2^{-1}$	0011 1111 0100 0000 0000 0000 0000 0000	0x3F400000

- The value 2.5. Exactly the same as two except that the bit that represents 1/4 is set in the significand.

Value	Formula	Binary representation	Hexadecimal
2.5	$1.25 * 2^1$	0100 0000 0010 0000 0000 0000 0000 0000	0x40200000

- 1/10 is a repeating fraction in binary. The significand is a little less than 1.6, and the biased exponent says that 1.6 is to be divided by 16. (It's 011 1101 1 in binary, which is 123 in decimal.) The true exponent is 123 - 127 = -4, which means that the factor by which to multiply is 2^{-4} = 1/16. The stored significand is rounded up in the last bit in an attempt to represent the unrepresentable number as accurately as possible. (The reason that 1/10 and 1/100 aren't exactly representable in binary is similar to the reason that 1/3 isn't exactly representable in decimal.)

Value	Formula	Binary representation	Hexadecimal
0.1	$1.6 * 2^{-4}$	0011 1101 1100 1100 1100 1100 1100 1101	0x3DCCCCCD

- Zero is a special case. It uses the formula for the minimum possible representable positive value, which is all zeroes.

Value	Formula	Binary representation	Hexadecimal
0	$1 * 2^{-128}$	0000 0000 0000 0000 0000 0000 0000 0000	0x00000000

See also

Why Floating-Point Numbers May Lose Precision

Tips for Improving Time-Critical Code

Article • 02/07/2023

Writing fast code requires understanding all aspects of your application and how it interacts with the system. This article suggests alternatives to some of the more obvious coding techniques to help you ensure that the time-critical portions of your code perform satisfactorily.

To summarize, improving time-critical code requires that you:

- Know which parts of your program have to be fast.
- Know the size and speed of your code.
- Know the cost of new features.
- Know the minimum work needed to accomplish the job.

To gather information on the performance of your code, you can use the performance monitor (perfmon.exe).

Sections in this Article

- Cache Misses and Page Faults
- Sorting and Searching
- MFC and Class Libraries
- Shared Libraries
- Heaps
- Threads
- Small Working Set

Cache Misses and Page Faults

Missed cache hits, on both the internal and external cache, as well as page faults (going to secondary storage for program instructions and data) slow the performance of a program.

A CPU cache hit can cost your program 10-20 clock cycles. An external cache hit can cost 20-40 clock cycles. A page fault can cost a million clock cycles (assuming a processor that handles 500 million instructions/second and a time of 2 millisecond for a page fault). Therefore, it is in the best interest of program execution to write code that will reduce the number of missed cache hits and page faults.

One reason for slow programs is that they take more page faults or miss the cache more often than necessary. To avoid this problem, it's important to use data structures with good locality of reference, which means keeping related things together. Sometimes a data structure that looks great turns out to be horrible because of poor locality of reference, and sometimes the reverse is true. Here are two examples:

- Dynamically allocated linked lists can reduce program performance. When you search for an item, or when you traverse a list to the end, each skipped link could miss the cache or cause a page fault. A list implementation based on simple arrays might be faster because of better caching and fewer page faults. Even if you allow for the fact that the array would be harder to grow, it still might be faster.

- Hash tables that use dynamically allocated linked lists can degrade performance. By extension, hash tables that use dynamically allocated linked lists to store their contents might perform substantially worse. In fact, in the final analysis, a simple linear search through an array might actually be faster (depending on the circumstances). Use of an array-based hash table (so-called "closed hashing") is an often-overlooked implementation that frequently has superior performance.

Sorting and Searching

Sorting is inherently time consuming compared to many typical operations. The best way to avoid unnecessary slowdown is to avoid sorting at critical times. You may be able to:

- Defer sorting until a non-performance-critical time.

- Sort the data at an earlier, non-performance-critical time.

- Sort only the part of the data that truly needs sorting.

Sometimes, you can build the list in sorted order. Be careful, because if you need to insert data in sorted order, you may require a more complicated data structure with poor locality of reference, leading to cache misses and page faults. There's no approach that works in all cases. Try several approaches and measure the differences.

Here are some general tips for sorting:

- Use a stock sort to minimize bugs.
- Any work you can do beforehand to reduce the complexity of the sort is worthwhile. If a one-time pass over your data simplifies the comparisons and reduces the sort from O(n log n) to O(n), you'll almost certainly come out ahead.
- Think about the locality of reference of the sort algorithm and the data you expect it to run on.

There are fewer alternatives for searches than for sorting. If the search is time-critical, a binary search or hash table lookup is almost always best, but as with sorting, you must keep locality in mind. A linear search through a small array can be faster than a binary search through a data structure with many pointers that causes page faults or cache misses.

MFC and Class Libraries

The Microsoft Foundation Classes (MFC) can greatly simplify writing code. When writing time-critical code, you should be aware of the overhead inherent in some of the classes. Examine the MFC code that your time-critical code uses to see if it meets your performance requirements. The following list identifies MFC classes and functions you should be aware of:

- `CString` MFC calls the C run-time library to allocate memory for a CString dynamically. Generally speaking, `CString` is as efficient as any other dynamically allocated string. As with any dynamically allocated string, it has the overhead of dynamic allocation and release. Often, a simple `char` array on the stack can serve the same purpose and is faster. Don't use a `CString` to store a constant string. Use `const char *` instead. Any operation you perform with a `CString` object has some overhead. Using the run-time library string functions may be faster.
- `CArray` A CArray provides flexibility that a regular array doesn't, but your program may not need that. If you know the specific limits for the array, you can use a global fixed array instead. If you use `CArray`, use `CArray::SetSize` to establish its size and specify the number of elements by which it grows when a reallocation is necessary. Otherwise, adding elements can cause your array to be frequently reallocated and copied, which is inefficient and can fragment memory. Also, if you insert an item into an array, `CArray` moves subsequent items in memory and may need to grow the array. These actions can cause cache misses and page faults. If you look through the code that MFC uses, you may see that you can write something more specific to your scenario to improve performance. Since `CArray` is

a template, for example, you might provide `CArray` specializations for specific types.

- `CList` CList is a doubly linked list, so element insertion is fast at the head, tail, and at a known position (`POSITION`) in the list. Looking up an element by value or index requires a sequential search, however, which can be slow if the list is long. If your code doesn't require a doubly linked list, you may want to reconsider using `CList`. Using a singly linked list saves the overhead of updating another pointer for all operations and the memory for that pointer. The extra memory isn't large, but it's another opportunity for cache misses or page faults.

- `IsKindOf` This function can generate many calls and may access memory in different data areas, leading to bad locality of reference. It's useful for a debug build (in an ASSERT call, for example), but try to avoid using it in a release build.

- `PreTranslateMessage` Use `PreTranslateMessage` when a particular tree of windows needs different keyboard accelerators or when you must insert message handling into the message pump. `PreTranslateMessage` alters MFC dispatch messages. If you override `PreTranslateMessage`, do so only at the level needed. For example, it isn't necessary to override `CMainFrame::PreTranslateMessage` if you're interested only in messages going to children of a particular view. Override `PreTranslateMessage` for the view class instead.

 Don't circumvent the normal dispatch path by using `PreTranslateMessage` to handle any message sent to any window. Use window procedures and MFC message maps for that purpose.

- `OnIdle` Idle events can occur at times you don't expect, such as between `WM_KEYDOWN` and `WM_KEYUP` events. Timers may be a more efficient way to trigger your code. Don't force `OnIdle` to be called repeatedly by generating false messages or by always returning `TRUE` from an override of `OnIdle`, which would never allow your thread to sleep. Again, a timer or a separate thread might be more appropriate.

Shared libraries

Code reuse is desirable. However, if you're going to use someone else's code, you should make sure you know exactly what it does in those cases where performance is critical to you. The best way to understand it is by stepping through the source code or by measuring with tools such as PView or Performance Monitor.

Heaps

Use multiple heaps with discretion. Additional heaps created with `HeapCreate` and `HeapAlloc` let you manage and then dispose of a related set of allocations. Don't commit too much memory. If you're using multiple heaps, pay special attention to the amount of memory that is initially committed.

Instead of multiple heaps, you can use helper functions to interface between your code and the default heap. Helper functions facilitate custom allocation strategies that can improve the performance of your application. For example, if you frequently perform small allocations, you may want to localize these allocations to one part of the default heap. You can allocate a large block of memory and then use a helper function to suballocate from that block. Then you won't have multiple heaps with unused memory, because the allocation is coming out of the default heap.

In some cases, however, using the default heap can reduce locality of reference. Use Process Viewer, Spy++, or Performance Monitor to measure the effects of moving objects from heap to heap.

Measure your heaps so you can account for every allocation on the heap. Use the C run-time debug heap routines to checkpoint and dump your heap. You can read the output into a spreadsheet program like Microsoft Excel and use pivot tables to view the results. Note the total number, size, and distribution of allocations. Compare these results with the size of working sets. Also look at the clustering of related-sized objects.

You can also use the performance counters to monitor memory usage.

Threads

For background tasks, effective idle handling of events may be faster than using threads. It's easier to understand locality of reference in a single-threaded program.

A good rule of thumb is to use a thread only if an operating system notification that you block on is at the root of the background work. Threads are the best solution in such a case because it's impractical to block a main thread on an event.

Threads also present communication problems. You must manage the communication link between your threads, with a list of messages or by allocating and using shared memory. Managing the communication link usually requires synchronization to avoid race conditions and deadlock problems. This complexity can easily turn into bugs and performance problems.

For more information, see Idle Loop Processing and Multithreading.

Small Working Set

Smaller working sets mean better locality of reference, fewer page faults, and more cache hits. The process working set is the closest metric the operating system directly provides for measuring locality of reference.

- To set the upper and lower limits of the working set, use SetProcessWorkingSetSize.
- To get the upper and lower limits of the working set, use GetProcessWorkingSetSize.
- To view the size of the working set, use Spy++.

See also

Optimizing Your Code

Using Function Name Without () Produces No Code

Article • 08/03/2021

When a function name declared in your program is used without parentheses, the compiler does not produce code. This occurs regardless of whether or not the function takes parameters because the compiler calculates the function address; however, because the function call operator "()" is not present, no call is made. This result is similar to the following:

```
// compile with /Wall to generate a warning
int a;
a;      // no code generated here either
```

In Visual C++, even using warning level 4 generates no diagnostic output. No warning is issued; no code is produced.

The sample code below compiles (with a warning) and links correctly without errors but produces no code in reference to `funcn( )`. For this to work correctly, add the function call operator "()".

```
#include <stdio.h>
void funcn();

int main() {
   funcn;      /* missing function call operator;
                  call will fail.  Use funcn() */
   }

void funcn() {
   printf("\nHello World\n");
}
```

See also

Optimizing Your Code

Optimization best practices

Article • 08/03/2021

This document describes some best practices for optimizing C++ programs in Visual Studio.

Compiler and Linker Options

Profile-guided optimization

Visual Studio supports *profile-guided optimization* (PGO). This optimization uses profile data from training executions of an instrumented version of an application to drive later optimization of the application. Using PGO can be time consuming, so it may not be something that every developer uses, but we do recommend using PGO for the final release build of a product. For more information, see Profile-Guided Optimizations.

In addition, *Whole Program Optimization* (also knows as Link Time Code Generation) and the `/O1` and `/O2` optimizations have been improved. In general, an application compiled with one of these options will be faster than the same application compiled with an earlier compiler.

For more information, see /GL (Whole Program Optimization) and /O1, /O2 (Minimize Size, Maximize Speed).

Which level of optimization to use

If at all possible, final release builds should be compiled with Profile Guided Optimizations. If it is not possible to build with PGO, whether due to insufficient infrastructure for running the instrumented builds or not having access to scenarios, then we suggest building with Whole Program Optimization.

The `/Gy` switch is also very useful. It generates a separate COMDAT for each function, giving the linker more flexibility when it comes to removing unreferenced COMDATs and COMDAT folding. The only downside to using `/Gy` is that it can cause issues when debugging. Therefore, it is generally recommended to use it. For more information, see /Gy (Enable Function-Level Linking).

For linking in 64-bit environments, it is recommended to use the `/OPT:REF,ICF` linker option, and in 32-bit environments, `/OPT:REF` is recommended. For more information, see /OPT (Optimizations).

It is also strongly recommended to generate debug symbols, even with optimized release builds. It doesn't affect the generated code, and it makes it a lot easier to debug your application, if need be.

Floating-point switches

The `/Op` compiler option has been removed, and the following four compiler options dealing with floating point optimizations have been added:

Option	Description
`/fp:precise`	This is the default recommendation and should be used in most cases.
`/fp:fast`	Recommended if performance is of the utmost importance, for example in games. This will result in the fastest performance.
`/fp:strict`	Recommended if precise floating-point exceptions and IEEE behavior is desired. This will result in the slowest performance.
`/fp:except[-]`	Can be used in conjunction with `/fp:strict` or `/fp:precise`, but not `/fp:fast`.

For more information, see /fp (Specify Floating-Point Behavior).

Optimization declspecs

In this section we will look at two declspecs that can be used in programs to help performance: `__declspec(restrict)` and `__declspec(noalias)`.

The `restrict` declspec can only be applied to function declarations that return a pointer, such as `__declspec(restrict) void *malloc(size_t size);`

The `restrict` declspec is used on functions that return unaliased pointers. This keyword is used for the C-Runtime Library implementation of `malloc` since it will never return a pointer value that is already in use in the current program (unless you are doing something illegal, such as using memory after it has been freed).

The `restrict` declspec gives the compiler more information for performing compiler optimizations. One of the hardest things for a compiler to determine is what pointers alias other pointers, and using this information greatly helps the compiler.

It is worth pointing out that this is a promise to the compiler, not something that the compiler will verify. If your program uses this `restrict` declspec inappropriately, your program may have incorrect behavior.

For more information, see restrict.

The `noalias` declspec is also applied only to functions, and indicates that the function is a semi-pure function. A semi-pure function is one that references or modifies only locals, arguments, and first-level indirections of arguments. This declspec is a promise to the compiler, and if the function references globals or second-level indirections of pointer arguments then the compiler may generate code that breaks the application.

For more information, see noalias.

Optimization pragmas

There are also several useful pragmas for helping optimize code. The first one we'll discuss is `#pragma optimize`:

```
C++

#pragma optimize("{opt-list}", on | off)
```

This pragma allows you to set a given optimization level on a function-by-function basis. This is ideal for those rare occasions where your application crashes when a given function is compiled with optimization. You can use this to turn off optimizations for a single function:

```
C++

#pragma optimize("", off)
int myFunc() {...}
#pragma optimize("", on)
```

For more information, see optimize.

Inlining is one of the most important optimizations that the compiler performs and here we talk about a couple of the pragmas that help modify this behavior.

`#pragma inline_recursion` is useful for specifying whether or not you want the application to be able to inline a recursive call. By default it is off. For shallow recursion of small functions you may to turn this on. For more information, see inline_recursion.

Another useful pragma for limiting the depth of inlining is `#pragma inline_depth`. This is typically useful in situations where you're trying to limit the size of a program or function. For more information, see inline_depth.

`__restrict` and `__assume`

There are a couple of keywords in Visual Studio that can help performance: __restrict and __assume.

First, it should be noted that `__restrict` and `__declspec(restrict)` are two different things. While they are somewhat related, their semantics are different. `__restrict` is a type qualifier, like `const` or `volatile`, but exclusively for pointer types.

A pointer that is modified with `__restrict` is referred to as a *__restrict pointer*. A __restrict pointer is a pointer that can only be accessed through the __restrict pointer. In other words, another pointer cannot be used to access the data pointed to by the __restrict pointer.

`__restrict` can be a powerful tool for the Microsoft C++ optimizer, but use it with great care. If used improperly, the optimizer might perform an optimization that would break your application.

With `__assume`, a developer can tell the compiler to make assumptions about the value of some variable.

For example `__assume(a < 5);` tells the optimizer that at that line of code the variable `a` is less than 5. Again this is a promise to the compiler. If `a` is actually 6 at this point in the program then the behavior of the program after the compiler has optimized may not be what you would expect. `__assume` is most useful prior to switch statements and/or conditional expressions.

There are some limitations to `__assume`. First, like `__restrict`, it is only a suggestion, so the compiler is free to ignore it. Also, `__assume` currently works only with variable inequalities against constants. It does not propagate symbolic inequalities, for example, assume(a < b).

Intrinsic support

Intrinsics are function calls where the compiler has intrinsic knowledge about the call, and rather than calling a function in a library, it emits code for that function. The header file <intrin.h> contains all of the available intrinsics for each of the supported hardware platforms.

Intrinsics give the programmer the ability to go deep into the code without having to use assembly. There are several benefits to using intrinsics:

- Your code is more portable. Several of the intrinsics are available on multiple CPU architectures.
- Your code is easier to read, since the code is still written in C/C++.
- Your code gets the benefit of compiler optimizations. As the compiler gets better, the code generation for the intrinsics improves.

For more information, see Compiler Intrinsics.

Exceptions

There is a performance hit associated with using exceptions. Some restrictions are introduced when using try blocks that inhibit the compiler from performing certain optimizations. On x86 platforms there is additional performance degradation from try blocks due to additional state information that must be generated during code execution. On the 64-bit platforms, try blocks do not degrade performance as much, but once an exception is thrown, the process of finding the handler and unwinding the stack can be expensive.

Therefore, it is recommended to avoid introducing try/catch blocks into code that does not really need it. If you must use exceptions, use synchronous exceptions if possible. For more information, see Structured Exception Handling (C/C++).

Lastly, throw exceptions for exceptional cases only. Using exceptions for general control flow will likely make performance suffer.

See also

- Optimizing Your Code

Profile-guided optimizations

Article • 10/18/2022

Profile-guided optimization (PGO) lets you optimize a whole executable file, where the optimizer uses data from test runs of the .exe or .dll file. The data represents the likely performance of the program in a production environment.

Profile-guided optimizations are only available for x86, x64, or ARM64 native targets. Profile-guided optimizations aren't available for executable files that run on the common language runtime. Even if you produce an assembly with mixed native and managed code (by using the **/clr** compiler option), you can't use profile-guided optimization on just the native code. If you attempt to build a project with these options set in the IDE, a build error results.

> **Note**
>
> Information that's gathered from profiling test runs overrides optimizations that would otherwise be in effect if you specify **/Ob**, **/Os**, or **/Ot**. For more information, see **/Ob (Inline Function Expansion)** and **/Os, /Ot (Favor Small Code, Favor Fast Code)**.

Steps to optimize your app

To use profile-guided optimization, follow these steps to optimize your app:

- Compile one or more source code files with /GL.

 Each module built with **/GL** can be examined during profile-guided optimization test runs to capture run-time behavior. Every module in a profile-guided optimization build doesn't have to be compiled with **/GL**. However, only those modules compiled with **/GL** are instrumented and later available for profile-guided optimizations.

- Link using /LTCG and /GENPROFILE or /FASTGENPROFILE.

 Using both **/LTCG** and **/GENPROFILE** or **/FASTGENPROFILE** creates a `.pgd` file when the instrumented app is run. After test-run data is added to the `.pgd` file, it can be used as input to the next link step (creating the optimized image). When specifying **/GENPROFILE**, you can optionally add a **PGD**=*filename* argument to specify a nondefault name or location for the `.pgd` file. The combination of **/LTCG**

and **/GENPROFILE** or **/FASTGENPROFILE** linker options replaces the deprecated **/LTCG:PGINSTRUMENT** linker option.

- Profile the application.

 Each time a profiled EXE session ends, or a profiled DLL is unloaded, a `appname!N.pgc` file is created. A `.pgc` file contains information about a particular application test run. *appname* is the name of your app, and *N* is a number starting with 1 that's incremented based on the number of other `appname!N.pgc` files in the directory. You can delete a `.pgc` file if the test run doesn't represent a scenario you want to optimize.

 During a test run, you can force closure of the currently open `.pgc` file and the creation of a new `.pgc` file with the pgosweep utility (for example, when the end of a test scenario doesn't coincide with application shutdown).

 Your application can also directly invoke a PGO function, PgoAutoSweep, to capture the profile data at the point of the call as a `.pgc` file. It can give you finer control over the code covered by the captured data in your `.pgc` files. For an example of how to use this function, see the PgoAutoSweep documentation.

 When you create your instrumented build, by default, data collection is done in non-thread-safe mode, which is faster but may be imprecise. By using the **EXACT** argument to **/GENPROFILE** or **/FASTGENPROFILE**, you can specify data collection in thread-safe mode, which is more precise, but slower. This option is also available if you set the deprecated PogoSafeMode environment variable, or the deprecated **/POGOSAFEMODE** linker option, when you create your instrumented build.

- Link using **/LTCG** and **/USEPROFILE**.

 Use both the **/LTCG** and /USEPROFILE linker options to create the optimized image. This step takes as input the `.pgd` file. When you specify **/USEPROFILE**, you can optionally add a **PGD=***filename* argument to specify a non-default name or location for the `.pgd` file. You can also specify this name by using the deprecated **/PGD** linker option. The combination of **/LTCG** and **/USEPROFILE** replaces the deprecated **/LTCG:PGOPTIMIZE** and **/LTCG:PGUPDATE** linker options.

It's even possible to create the optimized executable file and later determine that additional profiling would be useful to create a more optimized image. If the instrumented image and its `.pgd` file are available, you can do additional test runs and rebuild the optimized image with the newer `.pgd` file, by using the same **/LTCG** and **/USEPROFILE** linker options.

> ⓘ **Note**
>
> Both `.pgc` and `.pgd` files are binary file types. If stored in a source control system, avoid any automatic transformation that may be made to text files.

Optimizations performed by PGO

The profile-guided optimizations include these checks and improvements:

- **Inlining** - For example, if a function A frequently calls function B, and function B is relatively small, then profile-guided optimizations inline function B in function A.
- **Virtual Call Speculation** - If a virtual call, or other call through a function pointer, frequently targets a certain function, a profile-guided optimization can insert a conditionally executed direct call to the frequently targeted function, and the direct call can be inlined.
- **Register Allocation** - Optimization based on profile data results in better register allocation.
- **Basic Block Optimization** - Basic block optimization allows commonly executed basic blocks that temporally execute within a given frame to be placed in the same set of pages (locality). It minimizes the number of pages used, which minimizes memory overhead.
- **Size/Speed Optimization** - Functions where the program spends the most execution time can be optimized for speed.
- **Function Layout** - Based on the call graph and profiled caller/callee behavior, functions that tend to be along the same execution path are placed in the same section.
- **Conditional Branch Optimization** - With the value probes, profile-guided optimizations can find if a given value in a switch statement is used more often than other values. This value can then be pulled out of the switch statement. The same can be done with `if`...`else` instructions where the optimizer can order the `if`...`else` so that either the `if` or `else` block is placed first, depending on which block is more frequently true.
- **Dead Code Separation** - Code that isn't called during profiling is moved to a special section that's appended to the end of the set of sections. It effectively keeps this section out of the often-used pages.

- **EH Code Separation** - Because EH code is only exceptionally executed, it can often be moved to a separate section. It's moved when profile-guided optimizations can determine that the exceptions occur only on exceptional conditions.
- **Memory Intrinsics** - Whether to expand an intrinsic or not depends on whether it's called frequently. An intrinsic can also be optimized based on the block size of moves or copies.

Next steps

Read more about these environment variables, functions, and tools you can use in profile-guided optimizations:

Environment variables for profile-guided optimizations
These variables were used to specify run-time behavior of testing scenarios. They're now deprecated and replaced by new linker options. This document shows you how to move from the environment variables to the linker options.

PgoAutoSweep
A function you can add to your app to provide fine-grained `.pgc` file data capture control.

pgosweep
A command-line utility that writes all profile data to the `.pgc` file, closes the `.pgc` file, and opens a new `.pgc` file.

pgomgr
A command-line utility that adds profile data from one or more `.pgc` files to the `.pgd` file.

How to: Merge multiple PGO profiles into a single profile
Examples of **pgomgr** usage.

See also

Additional MSVC build tools

Environment Variables for Profile-Guided Optimizations

Article • 08/03/2021

There are three environment variables that affect test scenarios on an image created with **/LTCG:PGI** for profile-guided optimizations:

- **PogoSafeMode** specifies whether to use fast mode or safe mode for application profiling.
- **VCPROFILE_ALLOC_SCALE** adds additional memory for use by the profiler.
- **VCPROFILE_PATH** lets you specify the folder used for .pgc files.

The PogoSafeMode and VCPROFILE_ALLOC_SCALE environment variables are deprecated starting in Visual Studio 2015. The linker options /GENPROFILE or /FASTGENPROFILE and /USEPROFILE specify the same linker behavior as these environment variables.

PogoSafeMode

This environment variable is deprecated. Use the **EXACT** or **NOEXACT** arguments to **/GENPROFILE** or **/FASTGENPROFILE** to control this behavior.

Clear or set the **PogoSafeMode** environment variable to specify whether to use fast mode or safe mode for application profiling on x86 systems.

Profile-guided optimization (PGO) has two possible modes during the profiling phase: *fast mode* and *safe mode*. When profiling is in fast mode, it uses the **INC** instruction to increase data counters. The **INC** instruction is faster but is not thread-safe. When profiling is in safe mode, it uses the **LOCK INC** instruction to increase data counters. The **LOCK INC** instruction has the same functionality as the **INC** instruction has, and is thread-safe, but it is slower than the **INC** instruction.

By default, PGO profiling operates in fast mode. **PogoSafeMode** is only required if you want to use safe mode.

To run PGO profiling in safe mode, you must either use the environment variable **PogoSafeMode** or the linker switch **/PogoSafeMode**, depending on the system. If you are performing the profiling on an x64 computer, you must use the linker switch. If you are performing the profiling on an x86 computer, you may use the linker switch or set

the **PogoSafeMode** environment variable to any value before you start the optimization process.

PogoSafeMode syntax

> **set PogoSafeMode**[=*value*]

Set **PogoSafeMode** to any value to enable safe mode. Set without a value to clear a previous value and re-enable fast mode.

VCPROFILE_ALLOC_SCALE

This environment variable is deprecated. Use the **MEMMIN** and **MEMMAX** arguments to **/GENPROFILE** or **/FASTGENPROFILE** to control this behavior.

Modify the **VCPROFILE_ALLOC_SCALE** environment variable to change the amount of memory allocated to hold the profile data. In rare cases, there will not be enough memory available to support gathering profile data when running test scenarios. In those cases, you can increase the amount of memory by setting **VCPROFILE_ALLOC_SCALE**. If you receive an error message during a test run that indicates that you have insufficient memory, assign a larger value to **VCPROFILE_ALLOC_SCALE**, until the test runs complete with no out-of-memory errors.

VCPROFILE_ALLOC_SCALE syntax

> **set VCPROFILE_ALLOC_SCALE**[=*scale_value*]

The *scale_value* parameter is a scaling factor for the amount of memory you want for running test scenarios. The default is 1. For example, this command line sets the scale factor to 2:

```
set VCPROFILE_ALLOC_SCALE=2
```

VCPROFILE_PATH

Use the **VCPROFILE_PATH** environment variable to specify the directory to create .pgc files. By default, .pgc files are created in the same directory as the binary being profiled. However, if the absolute path of the binary does not exist, as may be the case when you run profile scenarios on a different machine from where the binary was built, you can set **VCPROFILE_PATH** to a path that exists on the target machine.

VCPROFILE_PATH syntax

> **set VCPROFILE_PATH**[=*path*]

Set the *path* parameter to the directory path in which to add .pgc files. For example, this command line sets the folder to C:\profile:

```
set VCPROFILE_PATH=c:\profile
```

See also

Profile-Guided Optimizations
/GENPROFILE and /FASTGENPROFILE
/USEPROFILE

PgoAutoSweep

Article • 08/03/2021

`PgoAutoSweep` saves the current profile counter information to a file, and then resets the counters. Use the function during profile-guided optimization training to write all profile data from the running program to a `.pgc` file for later use in the optimization build.

Syntax

C++

```
void PgoAutoSweep(const char* name);    // ANSI/MBCS
void PgoAutoSweep(const wchar_t* name); // UNICODE
```

Parameters

name
An identifying string for the saved `.pgc` file.

Remarks

You can call `PgoAutoSweep` from your application to save and reset the profile data at any point during application execution. In an instrumented build, `PgoAutoSweep` captures the current profiling data, saves it in a file, and resets the profile counters. It's the equivalent of calling the pgosweep command at a specific point in your executable. In an optimized build, `PgoAutoSweep` is a no-op.

The saved profile counter data is placed in a file named *base_name-name*!*value*.pgc, where *base_name* is the base name of the executable, *name* is the parameter passed to `PgoAutoSweep`, and *value* is a unique value, usually a monotonically increasing number, to prevent file name collisions.

The `.pgc` files created by `PgoAutoSweep` must be merged into a `.pgd` file to be used to create an optimized executable. You can use the pgomgr command to perform the merge.

You can pass the name of the merged `.pgd` file to the linker during the optimization build by using the **PGD=***filename* argument to the /USEPROFILE linker option, or by using the deprecated **/PGD** linker option. If you merge the `.pgc` files into a file named

base_name.pgd, you do not need to specify the filename on the command line, because the linker picks up this file name by default.

The `PgoAutoSweep` function maintains the thread-safety setting specified when the instrumented build is created. If you use the default setting or specify the **NOEXACT** argument to the /GENPROFILE or /FASTGENPROFILE linker option, calls to `PgoAutoSweep` are not thread-safe. The **EXACT** argument creates a thread-safe and more accurate, but slower, instrumented executable.

Requirements

Routine	Required header
`PgoAutoSweep`	<pgobootrun.h>

The executable must include the pgobootrun.lib file in the linked libraries. This file is included in your Visual Studio installation, in the VC libraries directory for each supported architecture.

Example

The example below uses `PgoAutoSweep` to create two `.pgc` files at different points during execution. The first contains data that describes the runtime behavior until `count` is equal to 3, and the second contains the data collected after this point until just before application termination.

C++

```
// pgoautosweep.cpp
// Compile by using: cl /c /GL /W4 /EHsc /O2 pgoautosweep.cpp
// Link to instrument: link /LTCG /genprofile pgobootrun.lib
pgoautosweep.obj
// Run to generate data: pgoautosweep
// Merge data by using command line pgomgr tool:
// pgomgr /merge pgoautosweep-func1!1.pgc pgoautosweep-func2!1.pgc
pgoautosweep.pgd
// Link to optimize: link /LTCG /useprofile pgobootrun.lib pgoautosweep.obj

#include <iostream>
#include <windows.h>
#include <pgobootrun.h>

void func2(int count)
{
    std::cout << "hello from func2 " << count << std::endl;
```

```
    Sleep(2000);
}

void func1(int count)
{
    std::cout << "hello from func1 " << count << std::endl;
    Sleep(2000);
}

int main()
{
    int count = 10;
    while (count--)
    {
        if (count < 3)
            func2(count);
        else
        {
            func1(count);
            if (count == 3)
            {
                PgoAutoSweep("func1");
            }
        }
    }
    PgoAutoSweep("func2");
}
```

In a developer command prompt, compile the code to an object file by using this command:

```
cl /c /GL /W4 /EHsc /O2 pgoautosweep.cpp
```

Then generate an instrumented build for training by using this command:

```
link /LTCG /genprofile pgobootrun.lib pgoautosweep.obj
```

Run the instrumented executable to capture the training data. The data output by the calls to `PgoAutoSweep` is saved in files named pgoautosweep-func1!1.pgc and pgoautosweep-func2!1.pgc. The output of the program should look like this as it runs:

Output

```
hello from func1 9
hello from func1 8
hello from func1 7
hello from func1 6
hello from func1 5
hello from func1 4
hello from func1 3
hello from func2 2
```

```
hello from func2 1
hello from func2 0
```

Merge the saved data into a profile training database by running the **pgomgr** command:

```
pgoautosweep-func1!1.pgc pgoautosweep-func2!1.pgc
```

The output of this command looks something like this:

Output

```
Microsoft (R) Profile Guided Optimization Manager 14.13.26128.0
Copyright (C) Microsoft Corporation. All rights reserved.

Merging pgoautosweep-func1!1.pgc
pgoautosweep-func1!1.pgc: Used  3.8% (22304 / 589824) of total space
reserved.  0.0% of the counts were dropped due to overflow.
Merging pgoautosweep-func2!1.pgc
pgoautosweep-func2!1.pgc: Used  3.8% (22424 / 589824) of total space
reserved.  0.0% of the counts were dropped due to overflow.
```

Now you can use this training data to generate an optimized build. Use this command to build the optimized executable:

```
link /LTCG /useprofile pgobootrun.lib pgoautosweep.obj
```

Output

```
Microsoft (R) Incremental Linker Version 14.13.26128.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Merging pgoautosweep!1.pgc
pgoautosweep!1.pgc: Used  3.9% (22904 / 589824) of total space reserved.
0.0% of the counts were dropped due to overflow.
  Reading PGD file 1: pgoautosweep.pgd
Generating code

0 of 0 ( 0.0%) original invalid call sites were matched.
0 new call sites were added.
294 of 294 (100.00%) profiled functions will be compiled for speed
348 of 1239 inline instances were from dead/cold paths
294 of 294 functions (100.0%) were optimized using profile data
16870 of 16870 instructions (100.0%) were optimized using profile data
Finished generating code
```

See also

Profile-Guided Optimizations

pgosweep

pgomgr

Article • 08/03/2021

Adds profile data from one or more .pgc files to the .pgd file.

Syntax

> **pgomgr** [*options*] *pgcfiles pgdfile*

Parameters

options
The following options can be specified to **pgomgr**:

- **/help** or **/?** Displays available **pgomgr** options.
- **/clear** Causes the .pgd file to be cleared of all profile information. You cannot specify a .pgc file when **/clear** is specified.
- **/detail** Displays detailed statistics, including flow graph coverage information.
- **/summary** Displays per-function statistics.
- **/unique** When used with **/summary**, causes decorated function names to display. The default, when **/unique** is not used, is for undecorated function names to be displayed.
- **/merge**[**:***n*] Causes the data in the .pgc file or files to be added to the .pgd file. The optional parameter, *n*, lets you specify that the data should be added *n* times. For example, if a scenario would commonly be done six times to reflect how often it is done by customers, you can do it once in a test run and add it to the .pgd file six times with **pgomgr /merge:6**.

pgcfiles
One or more .pgc files whose profile data you want to merge into the .pgd file. You can specify a single .pgc file or multiple .pgc files. If you do not specify any .pgc files, **pgomgr** merges all .pgc files whose filenames are the same as the .pgd file.

pgdfile
The .pgd file into which you are merging data from the .pgc file or files.

Remarks

> **Note**
>
> You can start this tool only from a Visual Studio developer command prompt. You cannot start it from a system command prompt or from File Explorer.

Example

This example command clears the myapp.pgd file of profile data:

```
pgomgr /clear myapp.pgd
```

This example command adds profile data in myapp1.pgc to the .pgd file three times:

```
pgomgr /merge:3 myapp1.pgc myapp.pgd
```

In this example, profile data from all myapp#.pgc files is added to the myapp.pgd file.

```
pgomgr -merge myapp1.pgd
```

See also

Profile-Guided Optimizations
PgoAutoSweep
pgosweep

pgosweep

Article • 08/03/2021

Used in profile-guided optimization to write all profile data from a running program to the PGC file.

Syntax

> `pgosweep` [*options*] *image pgcfile*

Parameters

options
(Optional) The valid values for *options* are:

- `/?` or `/help` displays the help message.
- `/reset` resets counts to zero after sweep. This behavior is the default.
- `/pid:n` only sweeps the specified PID, where *n* is the PID number.
- `/wait` waits for the specified PID to terminate before collecting counts.
- `/onlyzero` doesn't save a PGC file, only zero counts.
- `/pause` pauses count collection on the system.
- `/resume` resumes count collection on the system.
- `/noreset` preserves the count in the runtime data structures.

image
The full path of an EXE or DLL file that was created by using the /GENPROFILE, /FASTGENPROFILE, or /LTCG:PGINSTRUMENT option.

pgcfile
The PGC file where this command writes out the data counts.

Remarks

The `pgosweep` command works on programs that were built by using the /GENPROFILE or /FASTGENPROFILE option, or the deprecated /LTCG:PGINSTRUMENT option. It

interrupts a running program and writes the profile data to a new PGC file. By default, the command resets counts after each write operation. If you specify the `/noreset` option, the command will record the values, but not reset them in the running program. This option gives you duplicate data if you retrieve the profile data later.

An alternative use for `pgosweep` is to retrieve profile information just for the normal operation of the application. For example, you could run `pgosweep` shortly after you start the application and discard that file. This command would remove profile data associated with startup costs. Then, you can run `pgosweep` before ending the application. Now the collected data has profile information only from the time the user could interact with the program.

When you name a PGC file (by using the *pgcfile* parameter) you can use the standard format, which is *`appname!n.pgc`*. The *n* represents an increasing numeric value for each file. If you use this format, the compiler automatically finds this data in the `/LTCG` `/USEPROFILE` or `/LTCG:PGO` phase. If you don't use the standard format, you must use pgomgr to merge the PGC files.

> **Note**
>
> You can start this tool only from a Visual Studio developer command prompt. You can't start it from a system command prompt or from File Explorer.

For information on how to capture the profile data from within your executable, see PgoAutoSweep.

Example

In this example command, `pgosweep` writes the current profile information for *`myapp.exe`* to *`myapp!1.pgc`*.

```
pgosweep myapp.exe myapp!1.pgc
```

See also

Profile-Guided Optimizations
PgoAutoSweep

How to: Merge Multiple PGO Profiles into a Single Profile

Article • 08/03/2021

Profile-guided optimization (PGO) is a great tool for creating optimized binaries based on a scenario that is profiled. But what if you have an application that has several important, yet distinct scenarios? How do you create a single profile that PGO can use from several different scenarios? In Visual Studio, the PGO Manager, pgomgr.exe, does this job for you.

The syntax for merging profiles is:

```
pgomgr /merge[:num] [.pgc_files] .pgd_files
```

where `num` is an optional weight to use for the .pgc files added by this merge. Weights are commonly used if there are some scenarios that are more important than others or if there are scenarios that are to be run multiple times.

> **Note**
>
> The PGO Manager does not work with stale profile data. To merge a .pgc file into a .pgd file, the .pgc file must be generated by an executable which was created by the same link invocation that generated the .pgd file.

Examples

In this example, the PGO Manager adds pgcFile.pgc to pgdFile.pgd six times:

```
pgomgr /merge:6 pgcFile.pgc pgdFile.pgd
```

In this example, the PGO Manager adds pgcFile1.pgc and pgcFile2.pgc to pgdFile.pgd, two times for each .pgc file:

```
pgomgr /merge:2 pgcFile1.pgc pgcFile2.pgc pgdFile.pgd
```

If the PGO Manager is run without any .pgc file arguments, it searches the local directory for all .pgc files that have the same base name as the .pgd file followed by an exclamation mark (!) and then one or more arbitrary characters. For example, if the local directory has files test.pgd, test!1.pgc, test2.pgc, and test!hello.pgc, and the following

command is run from the local directory, then **pgomgr** merges test!1.pgc and test!hello.pgc into test.pgd.

```
pgomgr /merge test.pgd
```

See also

Profile-Guided Optimizations

Use the Microsoft C++ toolset from the command line

Article • 03/02/2023

You can build C and C++ applications on the command line by using tools that are included in Visual Studio. The Microsoft C++ (MSVC) compiler toolset is also downloadable as a standalone package. You don't need to install the Visual Studio IDE if you don't plan to use it.

> **Note**
>
> This article is about how to set up an environment to use the individual compilers, linkers, librarian, and other basic tools. The native project build system in Visual Studio, based on MSBuild, doesn't use the environment as described in this article. For more information on how to use MSBuild from the command line, see **MSBuild on the command line - C++**.

Download and install the tools

If you've installed Visual Studio and a C++ workload, you have all the command-line tools. For information on how to install C++ and Visual Studio, see Install C++ support in Visual Studio. If you only want the command-line toolset, download the Build Tools for Visual Studio. When you run the downloaded executable, it updates and runs the Visual Studio Installer. To install only the tools you need for C++ development, select the **Desktop development with C++** workload. You can select optional libraries and toolsets to include under **Installation details**. To build code by using the Visual Studio 2015, 2017, or 2019 toolsets, select the optional MSVC v140, v141, or v142 build tools. When you're satisfied with your selections, choose **Install**.

How to use the command-line tools

When you choose one of the C++ workloads in the Visual Studio Installer, it installs the Visual Studio *platform toolset*. A platform toolset has all the C and C++ tools for a specific Visual Studio version. The tools include the C/C++ compilers, linkers, assemblers, and other build tools, and matching libraries and header files. You can use all of these tools at the command line. They're also used internally by the Visual Studio IDE. There are separate x86-hosted and x64-hosted compilers and tools to build code

for x86, x64, ARM, and ARM64 targets. Each set of tools for a particular host and target build architecture is stored in its own directory.

To work correctly, the tools require several specific environment variables to be set. These variables are used to add the tools to the path, and to set the locations of include files, library files, and SDKs. To make it easy to set these environment variables, the installer creates customized *command files*, or batch files, during installation. You can run one of these command files to set a specific host and target build architecture, Windows SDK version, and platform toolset. For convenience, the installer also creates shortcuts in your Start menu. The shortcuts open developer command prompt windows by using these command files for specific combinations of host and target. These shortcuts ensure all the required environment variables are set and ready to use.

The required environment variables are specific to your installation and to the build architecture you choose. They also might be changed by product updates or upgrades. This variability is one reason why we recommend you use an installed command prompt shortcut or command file, instead of setting the environment variables yourself.

The toolsets, command files, and shortcuts installed depend on your computer processor and the options you selected during installation. The x86-hosted tools and cross tools that build x86 and x64 code are always installed. If you have 64-bit Windows, the x64-hosted tools and cross tools that build x86 and x64 code are also installed. If you choose the optional C++ Universal Windows Platform tools, then the x86 and x64 tools that build ARM and ARM64 code also get installed. Other workloads may install these and other tools.

Path and environment variables for command-line builds

The MSVC command-line tools use the `PATH`, `TMP`, `INCLUDE`, `LIB`, and `LIBPATH` environment variables, and also use other environment variables specific to your installed tools, platforms, and SDKs. Even a simple Visual Studio installation may set twenty or more environment variables. This complexity is why we strongly recommend that you use a developer command prompt shortcut or one of the customized command files. We don't recommend you set these variables in the Windows environment yourself.

To see which environment variables are set by a developer command prompt shortcut, you can use the `SET` command. Open a plain command prompt window and capture the output of the `SET` command for a baseline. Open a developer command prompt window and capture the output of the `SET` command for comparison. Use a diff tool

such as the one built into Visual Studio to highlight the environment variables set by the developer command prompt. For more information about the compiler and linker environment variables, see CL environment variables.

Developer command prompt shortcuts

The command prompt shortcuts are installed in a version-specific Visual Studio folder in your Windows Start menu. Here's a list of the base command prompt shortcuts and the build architectures they support:

- **Developer Command Prompt** - Sets the environment to use 32-bit, x86-native tools to build 32-bit, x86-native code.
- **x86 Native Tools Command Prompt** - Sets the environment to use 32-bit, x86-native tools to build 32-bit, x86-native code.
- **x64 Native Tools Command Prompt** - Sets the environment to use 64-bit, x64-native tools to build 64-bit, x64-native code.
- **x86_x64 Cross Tools Command Prompt** - Sets the environment to use 32-bit, x86-native tools to build 64-bit, x64-native code.
- **x64_x86 Cross Tools Command Prompt** - Sets the environment to use 64-bit, x64-native tools to build 32-bit, x86-native code.

The Start menu folder and shortcut names vary depending on the installed version of Visual Studio. If you set one, they also depend on the installation **Nickname**. For example, suppose you installed Visual Studio 2022, and you gave it a nickname of *Latest*. The developer command prompt shortcut is named **Developer Command Prompt for VS 2022 (Latest)**, in a folder named **Visual Studio 2022**.

> **Note**
>
> Several command-line tools or tool options may require Administrator permission. If you have permission issues when you use them, we recommend that you open the developer command prompt window by using the **Run as Administrator** option. Right-click to open the shortcut menu for the command prompt window, then choose **More**, **Run as administrator**.

To open a developer command prompt window

1. On the desktop, open the Windows **Start** menu. In Windows 11, choose the **All apps** button to open the list of installed apps. In Windows 10, the list is open to

the left. Scroll down the list to find and open the folder (not the app) for your version of Visual Studio, for example, **Visual Studio 2022**.

2. In the folder, choose the **Developer Command Prompt** for your version of Visual Studio. This shortcut starts a developer command prompt window that uses the default build architecture of 32-bit, x86-native tools to build 32-bit, x86-native code. If you prefer a non-default build architecture, choose one of the native or cross tools command prompts to specify the host and target architecture.

For an even faster way to open a developer command prompt, enter *developer command prompt* in the desktop search box. Then choose the result you want.

> **Note**
>
> By default, the current working directory in a developer command prompt is the root of your Visual Studio installation in the Program Files directory. This isn't an appropriate location for your code and projects. Change the current working directory to another location before you create a project. The IDE creates projects in your user directory, typically in *%USERPROFILE%\source\repos*.

Developer command file locations

If you prefer to set the build environment in an existing command prompt window, you can use one of the command files created by the installer. We recommend you set the environment in a new command prompt window. We don't recommend you later switch environments in the same command window.

The command file location depends on the version of Visual Studio you installed, and on choices you made during installation. For Visual Studio 2019, the typical installation location on a 64-bit system is in `\Program Files\Microsoft Visual Studio\2022\<edition>`. The `<edition>` may be Community, Professional, Enterprise, BuildTools, or another nickname you supplied.

The primary developer command prompt command file, `VsDevCmd.bat`, is located in the `Common7\Tools` subdirectory. When no parameters are specified, it sets the environment to use the x86-native tools to build 32-bit x86 code.

More command files are available to set up specific build architectures. The command files available depend on the Visual Studio workloads and options you've installed. In Visual Studio 2017 and Visual Studio 2019, you'll find them in the VC\Auxiliary\Build subdirectory.

These command files set default parameters and call `VsDevCmd.bat` to set up the specified build architecture environment. A typical installation may include these command files:

Command File	Host and Target architectures
`vcvars32.bat`	Use the 32-bit x86-native tools to build 32-bit x86 code.
`vcvars64.bat`	Use the 64-bit x64-native tools to build 64-bit x64 code.
`vcvarsx86_amd64.bat`	Use the 32-bit x86-native cross tools to build 64-bit x64 code.
`vcvarsamd64_x86.bat`	Use the 64-bit x64-native cross tools to build 32-bit x86 code.
`vcvarsx86_arm.bat`	Use the 32-bit x86-native cross tools to build ARM code.
`vcvarsamd64_arm.bat`	Use the 64-bit x64-native cross tools to build ARM code.
`vcvarsx86_arm64.bat`	Use the 32-bit x86-native cross tools to build ARM64 code.
`vcvarsamd64_arm64.bat`	Use the 64-bit x64-native cross tools to build ARM64 code.
`vcvarsall.bat`	Use parameters to specify the host and target architectures, Windows SDK, and platform choices. For a list of supported options, call by using a `/help` parameter.

⊗ Caution

The `vcvarsall.bat` file and other Visual Studio command files can vary from computer to computer. Do not replace a missing or damaged `vcvarsall.bat` file by using a file from another computer. Rerun the Visual Studio installer to replace the missing file.

The `vcvarsall.bat` file also varies from version to version. If the current version of Visual Studio is installed on a computer that also has an earlier version of Visual Studio, do not run `vcvarsall.bat` or another Visual Studio command file from different versions in the same command prompt window.

Use the developer tools in an existing command window

The simplest way to specify a particular build architecture in an existing command window is to use the `vcvarsall.bat` file. Use `vcvarsall.bat` to set environment variables to configure the command line for native 32-bit or 64-bit compilation. Arguments let

you specify cross-compilation to x86, x64, ARM, or ARM64 processors. You can target Microsoft Store, Universal Windows Platform, or Windows Desktop platforms. You can even specify which Windows SDK to use, and select the platform toolset version.

When used with no arguments, `vcvarsall.bat` configures the environment variables to use the current x86-native compiler for 32-bit Windows Desktop targets. You can add arguments to configure the environment to use any of the native or cross compiler tools. `vcvarsall.bat` displays an error message if you specify a configuration that's not installed, or not available on your computer.

`vcvarsall` syntax

> **`vcvarsall.bat`** [*`architecture`*] [*`platform_type`*] [*`winsdk_version`*] [**`-vcvars_ver=`***`vcversion`*] [*`spectre_mode`*]

`architecture`

This optional argument specifies the host and target architecture to use. If *architecture* isn't specified, the default build environment is used. These arguments are supported:

`architecture`	**Compiler**	**Host computer architecture**	**Build output (target) architecture**
`x86`	x86 32-bit native	x86, x64	x86
`x86_amd64` or `x86_x64`	x64 on x86 cross	x86, x64	x64
`x86_arm`	ARM on x86 cross	x86, x64	ARM
`x86_arm64`	ARM64 on x86 cross	x86, x64	ARM64
`amd64` or `x64`	x64 64-bit native	x64	x64
`amd64_x86` or `x64_x86`	x86 on x64 cross	x64	x86
`amd64_arm` or `x64_arm`	ARM on x64 cross	x64	ARM
`amd64_arm64` or `x64_arm64`	ARM64 on x64 cross	x64	ARM64

platform_type

This optional argument allows you to specify **store** or **uwp** as the platform type. By default, the environment is set to build desktop or console apps.

winsdk_version

Optionally specifies the version of the Windows SDK to use. By default, the latest installed Windows SDK is used. To specify the Windows SDK version, you can use a full Windows SDK number such as **10.0.10240.0**, or specify **8.1** to use the Windows 8.1 SDK.

vcversion

Optionally specifies the Visual Studio compiler toolset to use. By default, the environment is set to use the current Visual Studio compiler toolset.

Use **-vcvars_ver=14.2x.yyyyy** to specify a specific version of the Visual Studio 2019 compiler toolset.

Use **-vcvars_ver=14.29** to specify the latest version of the Visual Studio 2019 compiler toolset.

Use **-vcvars_ver=14.0** to specify the Visual Studio 2015 compiler toolset.

spectre_mode

Leave this parameter out to use libraries without Spectre mitigations. Use the value **spectre** to use libraries with Spectre mitigations.

To set up the build environment in an existing command prompt window

1. At the command prompt, use the CD command to change to the Visual Studio installation directory. Then, use CD again to change to the subdirectory that contains the configuration-specific command files. For Visual Studio 2019 and Visual Studio 2017, use the *VC\Auxiliary\Build* subdirectory. For Visual Studio 2015, use the *VC* subdirectory.

2. Enter the command for your preferred developer environment. For example, to build ARM code for UWP on a 64-bit platform, using the latest Windows SDK and Visual Studio compiler toolset, use this command line:

   ```
   vcvarsall.bat amd64_arm uwp
   ```

Create your own command prompt shortcut

Open the Properties dialog for a developer command prompt shortcut to see the command target used. For example, the target for the **x64 Native Tools Command Prompt for VS 2019** shortcut is something similar to:

```
%comspec% /k "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Build\vcvars64.bat"
```

The architecture-specific batch files set the *`architecture`* parameter and call *`vcvarsall.bat`*. You can pass the same options to these batch files as you would pass to *`vcvarsall.bat`*, or you can just call *`vcvarsall.bat`* directly. To specify parameters for your own command shortcut, add them to the end of the command in double-quotes. For example, here's a shortcut to build ARM code for UWP on a 64-bit platform, using the latest Windows SDK. To use an earlier compiler toolset, specify the version number. Use something like this command target in your shortcut:

```
%comspec% /k "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Build\vcvarsall.bat" amd64_arm uwp -vcvars_ver=14.29
```

Adjust the path to reflect your Visual Studio installation directory. The vcvarsall.bat file has additional information about specific version numbers.

Command-line tools

To build a C/C++ project at a command prompt, Visual Studio provides these command-line tools:

CL
Use the compiler (cl.exe) to compile and link source code files into apps, libraries, and DLLs.

Link
Use the linker (link.exe) to link compiled object files and libraries into apps and DLLs.

When you build on the command line, the F1 command isn't available for instant help. Instead, you can use a search engine to get information about warnings, errors, and messages. You can also download and use the offline help files. To use the search in Microsoft Learn, enter your query in the search box at the top of any article.

Command-line project management tools

By default, the Visual Studio IDE uses native project build systems based on MSBuild. You can invoke MSBuild directly to build projects without using the IDE. You can also use the `devenv` command to use Visual Studio to build projects and solutions. Visual Studio also supports build systems based on CMake or NMake.

MSBuild

Use MSBuild (msbuild.exe) and a project file (.vcxproj) to configure a build and invoke the toolset without loading the Visual Studio IDE. It's equivalent to running the **Build** project or **Build Solution** command in the Visual Studio IDE. MSBuild has advantages over the IDE when you build at the command line. You don't have to install the full IDE on all your build servers and build pipelines. You avoid the extra overhead of the IDE. MSBuild runs in containerized build environments, and supports a binary logger.

DEVENV

Use DEVENV (devenv.exe) combined with a command-line switch such as `/Build` or `/Clean` to execute certain build commands without displaying the Visual Studio IDE.

CMake

CMake (cmake.exe) is a cross-platform, open-source tool for defining build processes that run on multiple platforms. CMake can configure and control native build tools for its supported platforms, such as MSBuild and Make. For more information about CMake, see the CMake documentation.

NMAKE

Use NMAKE (nmake.exe) to build C++ projects by using a traditional makefile.

> **Note**
>
> Starting in Visual Studio 2019 version 16.5, MSBuild and DEVENV don't use the command-line environment to control the toolset and libraries used.

In this section

These articles show how to build apps on the command line, and describe how to customize the command-line build environment. Some show how to use 64-bit toolsets, and target x86, x64, ARM, and ARM64 platforms. They also describe use of the command-line build tools MSBuild and NMAKE.

Walkthrough: Compiling a native C++ program on the command line

Gives an example that shows how to create and compile a C++ program on the command line.

Walkthrough: Compile a C program on the command line

Describes how to compile a program written in the C programming language.

Walkthrough: Compiling a C++/CLI program on the command line

Describes how to create and compile a C++/CLI program that uses the .NET Framework.

Walkthrough: Compiling a C++/CX program on the command line

Describes how to create and compile a C++/CX program that uses the Windows Runtime.

NMAKE reference

Provides links to articles that describe the Microsoft Program Maintenance Utility (NMAKE.EXE).

MSBuild on the command line - C++

Provides links to articles that discuss how to use msbuild.exe from the command line.

Related sections

/MD, /MT, /LD (Use run-time library)

Describes how to use these compiler options to use a Debug or Release run-time library.

C/C++ compiler options

Provides links to articles that discuss the C and C++ compiler options and CL.exe.

MSVC linker options

Provides links to articles that discuss the linker options and LINK.exe.

Additional MSVC build tools

Provides links to the C/C++ build tools that are included in Visual Studio.

See also

Projects and build systems

Walkthrough: Compiling a Native C++ Program on the Command Line

Article • 02/08/2022

Visual Studio includes a command-line C and C++ compiler. You can use it to create everything from basic console apps to Universal Windows Platform apps, Desktop apps, device drivers, and .NET components.

In this walkthrough, you create a basic, "Hello, World"-style C++ program by using a text editor, and then compile it on the command line. If you'd like to try the Visual Studio IDE instead of using the command line, see Walkthrough: Working with Projects and Solutions (C++) or Using the Visual Studio IDE for C++ Desktop Development.

In this walkthrough, you can use your own C++ program instead of typing the one that's shown. Or, you can use a C++ code sample from another help article.

Prerequisites

To complete this walkthrough, you must have installed either Visual Studio and the optional **Desktop development with C++** workload, or the command-line Build Tools for Visual Studio.

Visual Studio is an *integrated development environment* (IDE). It supports a full-featured editor, resource managers, debuggers, and compilers for many languages and platforms. Versions available include the free Visual Studio Community edition, and all can support C and C++ development. For information on how to download and install Visual Studio, see Install C++ support in Visual Studio.

The Build Tools for Visual Studio installs only the command-line compilers, tools, and libraries you need to build C and C++ programs. It's perfect for build labs or classroom exercises and installs relatively quickly. To install only the command-line tools, look for Build Tools for Visual Studio on the Visual Studio Downloads page.

Before you can build a C or C++ program on the command line, verify that the tools are installed, and you can access them from the command line. Visual C++ has complex requirements for the command-line environment to find the tools, headers, and libraries it uses. **You can't use Visual C++ in a plain command prompt window** without doing some preparation. Fortunately, Visual C++ installs shortcuts for you to launch a developer command prompt that has the environment set up for command line builds. Unfortunately, the names of the developer command prompt shortcuts and where

they're located are different in almost every version of Visual C++ and on different versions of Windows. Your first walkthrough task is finding the right one to use.

> ⓘ **Note**
>
> A developer command prompt shortcut automatically sets the correct paths for the compiler and tools, and for any required headers and libraries. You must set these environment values yourself if you use a regular **Command Prompt** window. For more information, see **Use the MSVC toolset from the command line**. We recommend you use a developer command prompt shortcut instead of building your own.

Open a developer command prompt

1. If you have installed Visual Studio 2017 or later on Windows 10 or later, open the Start menu and choose **All apps**. Scroll down and open the **Visual Studio** folder (not the Visual Studio application). Choose **Developer Command Prompt for VS** to open the command prompt window.

 If you have installed Microsoft Visual C++ Build Tools 2015 on Windows 10 or later, open the **Start** menu and choose **All apps**. Scroll down and open the **Visual C++ Build Tools** folder. Choose **Visual C++ 2015 x86 Native Tools Command Prompt** to open the command prompt window.

 You can also use the Windows search function to search for "developer command prompt" and choose one that matches your installed version of Visual Studio. Use the shortcut to open the command prompt window.

2. Next, verify that the Visual C++ developer command prompt is set up correctly. In the command prompt window, enter `cl` and verify that the output looks something like this:

 Output

   ```
   C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise>cl
   Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
   Copyright (C) Microsoft Corporation.  All rights reserved.

   usage: cl [ option... ] filename... [ /link linkoption... ]
   ```

 There may be differences in the current directory or version numbers. These values depend on the version of Visual C++ and any updates installed. If the above

output is similar to what you see, then you're ready to build C or C++ programs at the command line.

> **Note**
>
> If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104 when you run the `cl` command, then either you are not using a developer command prompt, or something is wrong with your installation of Visual C++. You must fix this issue before you can continue.

If you can't find the developer command prompt shortcut, or if you get an error message when you enter `cl`, then your Visual C++ installation may have a problem. Try reinstalling the Visual C++ component in Visual Studio, or reinstall the Microsoft Visual C++ Build Tools. Don't go on to the next section until the `cl` command works. For more information about installing and troubleshooting Visual C++, see Install Visual Studio.

> **Note**
>
> Depending on the version of Windows on the computer and the system security configuration, you might have to right-click to open the shortcut menu for the developer command prompt shortcut and then choose **Run as administrator** to successfully build and run the program that you create by following this walkthrough.

Create a Visual C++ source file and compile it on the command line

1. In the developer command prompt window, enter `md c:\hello` to create a directory, and then enter `cd c:\hello` to change to that directory. This directory is where both your source file and the compiled program get created.

2. Enter `notepad hello.cpp` in the command prompt window.

 Choose **Yes** when Notepad prompts you to create a new file. This step opens a blank Notepad window, ready for you to enter your code in a file named hello.cpp.

3. In Notepad, enter the following lines of code:

   ```
   ```

C++

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, world, from Visual C++!" << endl;
}
```

This code is a simple program that will write one line of text on the screen and then exit. To minimize errors, copy this code and paste it into Notepad.

4. Save your work! In Notepad, on the **File** menu, choose **Save**.

 Congratulations, you've created a C++ source file, hello.cpp, that is ready to compile.

5. Switch back to the developer command prompt window. Enter `dir` at the command prompt to list the contents of the c:\hello directory. You should see the source file hello.cpp in the directory listing, which looks something like:

Output

```
c:\hello>dir
 Volume in drive C has no label.
 Volume Serial Number is CC62-6545

 Directory of c:\hello

05/24/2016  05:36 PM    <DIR>          .
05/24/2016  05:36 PM    <DIR>          ..
05/24/2016  05:37 PM               115 hello.cpp
               1 File(s)            115 bytes
               2 Dir(s)  571,343,446,016 bytes free
```

The dates and other details will differ on your computer.

> **Note**
>
> If you don't see your source code file, *`hello.cpp`*, make sure the current working directory in your command prompt is the *`C:\hello`* directory you created. Also make sure that this is the directory where you saved your source file. And make sure that you saved the source code with a *`.cpp`* file name extension, not a *`.txt`* extension. Your source file gets saved in the current directory as a *`.cpp`* file automatically if you open Notepad at the command

prompt by using the `notepad hello.cpp` command. Notepad's behavior is different if you open it another way: By default, Notepad appends a `.txt` extension to new files when you save them. It also defaults to saving files in your *Documents* directory. To save your file with a `.cpp` extension in Notepad, choose **File** > **Save As**. In the **Save As** dialog, navigate to your `C:\hello` folder in the directory tree view control. Then use the **Save as type** dropdown control to select **All Files (*.*)**. Enter `hello.cpp` in the **File name** edit control, and then choose **Save** to save the file.

6. At the developer command prompt, enter `cl /EHsc hello.cpp` to compile your program.

 The cl.exe compiler generates an .obj file that contains the compiled code, and then runs the linker to create an executable program named hello.exe. This name appears in the lines of output information that the compiler displays. The output of the compiler should look something like:

 Output

   ```
   c:\hello>cl /EHsc hello.cpp
   Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
   Copyright (C) Microsoft Corporation.  All rights reserved.

   hello.cpp
   Microsoft (R) Incremental Linker Version 14.10.25017.0
   Copyright (C) Microsoft Corporation.  All rights reserved.

   /out:hello.exe
   hello.obj
   ```

 Note

 If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104, your developer command prompt is not set up correctly. For information on how to fix this issue, go back to the **Open a developer command prompt** section.

 Note

 If you get a different compiler or linker error or warning, review your source code to correct any errors, then save it and run the compiler again. For

information about specific errors, use the search box to look for the error number.

7. To run the hello.exe program, at the command prompt, enter `hello`.

 The program displays this text and exits:

   ```
   Output

   Hello, world, from Visual C++!
   ```

 Congratulations, you've compiled and run a C++ program by using the command-line tools.

Next steps

This "Hello, World" example is about as simple as a C++ program can get. Real world programs usually have header files, more source files, and link to libraries.

You can use the steps in this walkthrough to build your own C++ code instead of typing the sample code shown. These steps also let you build many C++ code sample programs that you find elsewhere. You can put your source code and build your apps in any writeable directory. By default, the Visual Studio IDE creates projects in your user folder, in a *source\repos* subfolder. Older versions may put projects in a *Documents\Visual Studio <version>\Projects* folder.

To compile a program that has additional source code files, enter them all on the command line, like:

```
cl /EHsc file1.cpp file2.cpp file3.cpp
```

The `/EHsc` command-line option instructs the compiler to enable standard C++ exception handling behavior. Without it, thrown exceptions can result in undestroyed objects and resource leaks. For more information, see /EH (Exception Handling Model).

When you supply additional source files, the compiler uses the first input file to create the program name. In this case, it outputs a program called file1.exe. To change the name to program1.exe, add an /out linker option:

```
cl /EHsc file1.cpp file2.cpp file3.cpp /link /out:program1.exe
```

And to catch more programming mistakes automatically, we recommend you compile by using either the /W3 or /W4 warning level option:

```
cl /W4 /EHsc file1.cpp file2.cpp file3.cpp /link /out:program1.exe
```

The compiler, cl.exe, has many more options. You can apply them to build, optimize, debug, and analyze your code. For a quick list, enter `cl /?` at the developer command prompt. You can also compile and link separately and apply linker options in more complex build scenarios. For more information on compiler and linker options and usage, see C/C++ Building Reference.

You can use NMAKE and makefiles, MSBuild and project files, or CMake, to configure and build more complex projects on the command line. For more information on using these tools, see NMAKE Reference, MSBuild, and CMake projects in Visual Studio.

The C and C++ languages are similar, but not the same. The MSVC compiler uses a simple rule to determine which language to use when it compiles your code. By default, the MSVC compiler treats files that end in `.c` as C source code, and files that end in `.cpp` as C++ source code. To force the compiler to treat all files as C++ independent of file name extension, use the /TP compiler option.

The MSVC compiler includes a C Runtime Library (CRT) that conforms to the ISO C99 standard, with minor exceptions. Portable code generally compiles and runs as expected. Certain obsolete library functions, and several POSIX function names, are deprecated by the MSVC compiler. The functions are supported, but the preferred names have changed. For more information, see Security Features in the CRT and Compiler Warning (level 3) C4996.

See also

C++ Language Reference
Projects and build systems
MSVC Compiler Options

Walkthrough: Compile a C program on the command line

Article • 05/10/2022

The Visual Studio build tools include a C compiler that you can use to create everything from basic console programs to full Windows Desktop applications, mobile apps, and more. Microsoft C/C++ (MSVC) is a C and C++ compiler that, in its latest versions, conforms to some of the latest C language standards, including C11 and C17.

This walkthrough shows how to create a basic, "Hello, World"-style C program by using a text editor, and then compile it on the command line. If you'd rather work in C++ on the command line, see Walkthrough: Compiling a Native C++ Program on the Command Line. If you'd like to try the Visual Studio IDE instead of using the command line, see Walkthrough: Working with Projects and Solutions (C++) or Using the Visual Studio IDE for C++ Desktop Development.

Prerequisites

To complete this walkthrough, you must have installed either Visual Studio or the Build Tools for Visual Studio and the optional Desktop development with C++ workload.

Visual Studio is a powerful integrated development environment that supports a full-featured editor, resource managers, debuggers, and compilers for many languages and platforms. For information on these features and how to download and install Visual Studio, including the free Visual Studio Community edition, see Install Visual Studio.

The Build Tools for Visual Studio version of Visual Studio installs only the command-line toolset, the compilers, tools, and libraries you need to build C and C++ programs. It's perfect for build labs or classroom exercises and installs relatively quickly. To install only the command-line toolset, download Build Tools for Visual Studio from the Visual Studio downloads page and run the installer. In the Visual Studio installer, select the **Desktop development with C++** workload (in older versions of Visual Studio, select the **C++ build tools** workload), and choose **Install**.

When you've installed the tools, there's another tool you'll use to build a C or C++ program on the command line. MSVC has complex requirements for the command-line environment to find the tools, headers, and libraries it uses. **You can't use MSVC in a plain command prompt window** without some preparation. You need a *developer command prompt* window, which is a regular command prompt window that has all the required environment variables set. Fortunately, Visual Studio installs shortcuts for you

to launch developer command prompts that have the environment set up for command line builds. Unfortunately, the names of the developer command prompt shortcuts and where they're located are different in almost every version of Visual Studio and on different versions of Windows. Your first walkthrough task is to find the right shortcut to use.

> **Note**
>
> A developer command prompt shortcut automatically sets the correct paths for the compiler and tools, and for any required headers and libraries. Some of these values are different for each build configuration. You must set these environment values yourself if you don't use one of the shortcuts. For more information, see **Use the MSVC toolset from the command line**. Because the build environment is complex, we strongly recommend you use a developer command prompt shortcut instead of building your own.

These instructions vary depending on which version of Visual Studio you're using. To see the documentation for your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

Open a developer command prompt in Visual Studio 2022

If you've installed Visual Studio 2022 on Windows 10 or later, open the Start menu, and choose **All apps**. Then, scroll down and open the **Visual Studio 2022** folder (not the Visual Studio 2022 app). Choose **Developer Command Prompt for VS 2022** to open the command prompt window.

If you're using a different version of Windows, look in your Start menu or Start page for a Visual Studio tools folder that contains a developer command prompt shortcut. You can also use the Windows search function to search for "developer command prompt" and choose one that matches your installed version of Visual Studio. Use the shortcut to open the command prompt window.

Next, verify that the developer command prompt is set up correctly. In the command prompt window, enter `cl` (or `CL`, case doesn't matter for the compiler name, but it does matter for compiler options). The output should look something like this:

```
Output
```

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise>cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]
```

There may be differences in the current directory or version numbers, depending on the version of Visual Studio and any updates installed. If the above output is similar to what you see, then you're ready to build C or C++ programs at the command line.

> **Note**
>
> If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104 when you run the **cl** command, then either you are not using a developer command prompt, or something is wrong with your installation of Visual Studio. You must fix this issue before you can continue.

If you can't find the developer command prompt shortcut, or if you get an error message when you enter `cl`, then your Visual Studio installation may have a problem. If you're using Visual Studio 2017 or later, try reinstalling the **Desktop development with C++** workload in the Visual Studio installer. For details, see Install C++ support in Visual Studio. Or, reinstall the Build Tools from the Visual Studio downloads page. Don't go on to the next section until the `cl` command works. For more information about installing and troubleshooting Visual Studio, see Install Visual Studio.

> **Note**
>
> Depending on the version of Windows on the computer and the system security configuration, you might have to right-click to open the shortcut menu for the developer command prompt shortcut and then choose **Run as Administrator** to successfully build and run the program that you create by following this walkthrough.

Create a C source file and compile it on the command line

1. In the developer command prompt window, enter `cd c:\` to change the current working directory to the root of your C: drive. Next, enter `md c:\hello` to create a

directory, and then enter `cd c:\hello` to change to that directory. This directory will hold your source file and the compiled program.

2. Enter `notepad hello.c` at the developer command prompt. In the Notepad alert dialog that pops up, choose **Yes** to create a new *`hello.c`* file in your working directory.

3. In Notepad, enter the following lines of code:

C

```c
#include <stdio.h>

int main()
{
    printf("Hello, World! This is a native C program compiled on the
command line.\n");
    return 0;
}
```

4. On the Notepad menu bar, choose **File** > **Save** to save *`hello.c`* in your working directory.

5. Switch back to the developer command prompt window. Enter `dir` at the command prompt to list the contents of the *`c:\hello`* directory. You should see the source file *`hello.c`* in the directory listing, which looks something like:

Output

```
C:\hello>dir
 Volume in drive C has no label.
 Volume Serial Number is CC62-6545

 Directory of C:\hello

10/02/2017  03:46 PM    <DIR>          .
10/02/2017  03:46 PM    <DIR>          ..
10/02/2017  03:36 PM               143 hello.c
               1 File(s)            143 bytes
               2 Dir(s)  514,900,566,016 bytes free
```

The dates and other details will differ on your computer. If you don't see your source code file, *`hello.c`*, make sure you've changed to the *`c:\hello`* directory you created, and in Notepad, make sure that you saved your source file in this

directory. Also make sure that you saved the source code with a `.c` file name extension, not a `.txt` extension.

6. To compile your program, enter `cl hello.c` at the developer command prompt.

 You can see the executable program name, hello.exe, in the lines of output information that the compiler displays:

 Output

   ```
   c:\hello>cl hello.c
   Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
   Copyright (C) Microsoft Corporation.  All rights reserved.

   hello.c
   Microsoft (R) Incremental Linker Version 14.10.25017.0
   Copyright (C) Microsoft Corporation.  All rights reserved.

   /out:hello.exe
   hello.obj
   ```

 > **Note**
 >
 > If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104, your developer command prompt is not set up correctly. For information on how to fix this issue, go back to the **Open a developer command prompt** section.
 >
 > If you get a different compiler or linker error or warning, review your source code to correct any errors, then save it and run the compiler again. For information about specific errors, use the search box at the top of this page to look for the error number.

7. To run your program, enter `hello` at the command prompt.

 The program displays this text and then exits:

 Output

   ```
   Hello, World! This is a native C program compiled on the command line.
   ```

 Congratulations, you've compiled and run a C program by using the command line.

Next steps

This "Hello, World" example is about as basic as a C program can get. Real world programs have header files and more source files, link in libraries, and do useful work.

You can use the steps in this walkthrough to build your own C code instead of typing the sample code shown. You can also build many C code sample programs that you find elsewhere. To compile a program that has more source code files, enter them all on the command line:

```
cl file1.c file2.c file3.c
```

The compiler outputs a program called *`file1.exe`*. To change the name to *`program1.exe`*, add an /out linker option:

```
cl file1.c file2.c file3.c /link /out:program1.exe
```

And to catch more programming mistakes automatically, we recommend you compile by using either the /W3 or /W4 warning level option:

```
cl /W4 file1.c file2.c file3.c /link /out:program1.exe
```

The compiler, cl.exe, has many more options you can apply to build, optimize, debug, and analyze your code. For a quick list, enter `cl /?` at the developer command prompt. You can also compile and link separately and apply linker options in more complex build scenarios. For more information on compiler and linker options and usage, see C/C++ Building Reference.

You can use NMAKE and makefiles, or MSBuild and project files to configure and build more complex projects on the command line. For more information on using these tools, see NMAKE Reference and MSBuild.

The C and C++ languages are similar, but not the same. The Microsoft C/C++ compiler (MSVC) uses a basic rule to determine which language to use when it compiles your code. By default, the MSVC compiler treats all files that end in *`.c`* as C source code, and all files that end in *`.cpp`* as C++ source code. To force the compiler to treat all files as C no matter the file name extension, use the /TC compiler option.

By default, MSVC is compatible with the ANSI C89 and ISO C99 standards, but not strictly conforming. In most cases, portable C code will compile and run as expected. The compiler provides optional support for the changes in ISO C11/C17. To compile with C11/C17 support, use the compiler flag **`/std:c11`** or **`/std:c17`**. C11/C17 support requires Windows SDK 10.0.20201.0 or later. Windows SDK 10.0.22000.0 or later is

recommended. You can download the latest SDK from the Windows SDK page. For more information, and instructions on how to install and use this SDK for C development, see Install C11 and C17 support in Visual Studio.

Certain library functions and POSIX function names are deprecated by MSVC. The functions are supported, but the preferred names have changed. For more information, see Security Features in the CRT and Compiler Warning (level 3) C4996.

See also

Walkthrough: Creating a Standard C++ Program (C++)
C Language Reference
Projects and build systems
Compatibility

Walkthrough: Compiling a C++/CLI Program on the Command Line

Article • 02/24/2023

You can create Visual C++ programs that target the Common Language Runtime (CLR) and use the .NET Framework, and build them on the command line. Visual C++ supports the C++/CLI programming language, which has additional types and operators to target the .NET programming model. For general information about the C++/CLI language, see .NET Programming with C++/CLI (Visual C++).

In this walkthrough, you use a text editor to create a basic C++/CLI program, and then compile it on the command line. (You can use your own C++/CLI program instead of typing the one that's shown, or you can use a C++/CLI code sample from another help article. This technique is useful for building and testing small modules that have no UI elements.)

Prerequisites

You understand the fundamentals of the C++ language.

Compiling a C++/CLI Program

The following steps show how to compile a C++/CLI console application that uses .NET Framework classes.

To enable compilation for C++/CLI, you must use the /clr compiler option. The MSVC compiler generates an .exe file that contains MSIL code—or mixed MSIL and native code—and links to the required .NET Framework libraries.

To compile a C++/CLI application on the command line

1. Open a **Developer Command Prompt** window. For specific instructions, see To open a developer command prompt window.

 Administrator credentials may be required to successfully compile the code, depending on the computer's operating system and configuration. To run the command prompt window as an administrator, right-click to open the shortcut menu for the command prompt and then choose **More** > **Run as administrator**.

2. Change the current working directory in the command prompt window to a directory you can write to, such as your Documents directory.

3. At the command prompt, enter `notepad basicclr.cpp`.

 Choose **Yes** when you're prompted to create a file.

4. In Notepad, enter these lines:

 C++

   ```
   int main()
   {
       System::Console::WriteLine("This is a C++/CLI program.");
   }
   ```

5. On the menu bar, choose **File** > **Save**.

 You've created a Visual C++ source file that uses a .NET Framework class (Console) in the System namespace.

6. At the command prompt, enter `cl /clr basicclr.cpp`. The cl.exe compiler compiles the source code into an .obj file that contains MSIL, and then runs the linker to generate an executable program named basicclr.exe.

7. To run the basicclr.exe program, at the command prompt, enter `basicclr`.

 The program displays this text and exits:

 Output

   ```
   This is a C++/CLI program.
   ```

See also

C++ Language Reference
Projects and build systems
MSVC Compiler Options

Walkthrough: Compiling a C++/CX Program on the Command Line

Article • 03/01/2023

> **Note**
>
> For new UWP apps and components, we recommend that you use **C++/WinRT**, a standard C++17 language projection for Windows Runtime APIs. C++/WinRT is available in the Windows SDK from version 1803 (10.0.17134.0) onward. C++/WinRT is implemented entirely in header files, and is designed to provide you with first-class access to the modern Windows API.

The Microsoft C++ compiler (MSVC) supports C++ component extensions (C++/CX), which has additional types and operators to target the Windows Runtime programming model. You can use C++/CX to build apps for Universal Windows Platform (UWP), and Windows desktop. For more information, see A Tour of C++/CX and Component Extensions for Runtime Platforms.

In this walkthrough, you use a text editor to create a basic C++/CX program, and then compile it on the command line. (You can use your own C++/CX program instead of typing the one that's shown, or you can use a C++/CX code sample from another help article. This technique is useful for building and testing small modules that have no UI elements.)

> **Note**
>
> You can also use the Visual Studio IDE to compile C++/CX programs. Because the IDE includes design, debugging, emulation, and deployment support that isn't available on the command line, we recommend that you use the IDE to build Universal Windows Platform (UWP) apps. For more information, see **Create a UWP app in C++**.

Prerequisites

You understand the fundamentals of the C++ language.

Compiling a C++/CX Program

To enable compilation for C++/CX, you must use the /ZW compiler option. The MSVC compiler generates an .exe file that targets the Windows Runtime, and links to the required libraries.

To compile a C++/CX application on the command line

1. Open a **Developer Command Prompt** window. For specific instructions, see To open a developer command prompt window.

 Administrator credentials may be required to successfully compile the code, depending on the computer's operating system and configuration. To run the command prompt window as an administrator, right-click to open the shortcut menu for the command prompt and then choose **More** > **Run as administrator**.

2. Change the current working directory in the command prompt window to a directory you can write to, such as your Documents directory.

3. At the command prompt, enter **notepad basiccx.cpp**.

 Choose **Yes** when you're prompted to create a file.

4. In Notepad, enter these lines:

 C++

   ```
   using namespace Platform;

   int main(Platform::Array<Platform::String^>^ args)
   {
       Platform::Details::Console::WriteLine("This is a C++/CX program.");
   }
   ```

5. On the menu bar, choose **File** > **Save**.

 You've created a C++ source file that uses the Windows Runtime Platform namespace namespace.

6. At the command prompt, enter `cl /EHsc /ZW basiccx.cpp /link /SUBSYSTEM:CONSOLE`. The `cl.exe` compiler compiles the source code into an `.obj` file, and then runs the linker to generate an executable program named basiccx.exe. The /EHsc compiler option specifies the C++ exception-handling model, and the /link flag specifies a console application.

7. To run the basiccx.exe program, at the command prompt, enter **basiccx**.

The program displays this text and exits:

Output

```
This is a C++/CX program.
```

See also

Projects and build systems
MSVC Compiler Options

MSBuild on the command line - C++

Article • 08/03/2021

In general, we recommend that you use Visual Studio to set project properties and invoke the MSBuild system. However, you can use the **MSBuild** tool directly from the command prompt. The build process is controlled by the information in a project file (.vcxproj) that you can create and edit. The project file specifies build options based on build stages, conditions, and events. In addition, you can specify zero or more command-line *options* arguments.

> **msbuild.exe** [*project_file*] [*options*]

Use the **/target** (or **/t**) and **/property** (or **/p**) command-line options to override specific properties and targets that are specified in the project file.

An essential function of the project file is to specify a *target*, which is a particular operation applied to your project, and the inputs and outputs that are required to perform that operation. A project file can specify one or more targets, which can include a default target.

Each target consists of a sequence of one or more *tasks*. Each task is represented by a .NET Framework class that contains one executable command. For example, the CL task contains the cl.exe command.

A *task parameter* is a property of the class task and typically represents a command-line option of the executable command. For example, the `FavorSizeOrSpeed` parameter of the `CL` task corresponds to the **/Os** and **/Ot** compiler options.

Additional task parameters support the MSBuild infrastructure. For example, the `Sources` task parameter specifies a set of tasks that can be consumed by other tasks. For more information about MSBuild tasks, see Task Reference.

Most tasks require inputs and outputs, such as file names, paths, and string, numeric, or Boolean parameters. For example, a common input is the name of a .cpp source file to compile. An important input parameter is a string that specifies the build configuration and platform, for example, "Debug|Win32". Inputs and outputs are specified by one or more user-defined XML `Item` elements contained in an `ItemGroup` element.

A project file can also specify user-defined *properties* and `ItemDefinitionGroup` *items*. Properties and items form name/value pairs that can be used as variables in the build. The name component of a pair defines a *macro*, and the value component declares the

macro value. A property macro is accessed by using $(*name*) notation, and an item macro is accessed by using %(*name*) notation.

Other XML elements in a project file can test macros, and then conditionally set the value of any macro or control the execution of the build. Macro names and literal strings can be concatenated to generate constructs such as a path and file name. On the command line, the **/property** option sets or overrides a project property. Items cannot be referenced on the command line.

The MSBuild system can conditionally execute a target before or after another target. Also, the system can build a target based on whether the files that the target consumes are newer than the files it emits.

For more information about MSBuild, see:

- MSBuild Overview of MSBuild concepts.
- MSBuild Reference Reference information about the MSBuild system.
- Project File Schema Reference Lists the MSBuild XML Schema elements, together with their attributes, and parent and child elements. Especially note the ItemGroup, PropertyGroup, Target, and Task elements.
- Command-Line Reference Describes the command-line arguments and options that you can use with msbuild.exe.
- Task Reference Describes MSBuild tasks. Especially note these tasks, which are specific to Visual C++: BscMake Task, CL Task, CPPClean Task, LIB Task, Link Task, MIDL Task, MT Task, RC Task, SetEnv Task, VCMessage Task

In This Section

Term	Definition
Walkthrough: Using MSBuild to Create a C++ Project	Demonstrates how to create a Visual Studio C++ project using **MSBuild**.
How to: Use Build Events in MSBuild Projects	Demonstrates how to specify an action that occurs at a particuler stage in the build: before the build starts; before the link step starts; or after the build ends.
How to: Add a Custom Build Step to MSBuild Projects	Demonstrates how to add a user-defined stage to the build sequence.

Term	Definition
How to: Add Custom Build Tools to MSBuild Projects	Demonstrates how to associate a build tool with a particular file.
How to: Integrate Custom Tools into the Project Properties	Demonstrates how to add options for a custom tool to the project properties.
How to: Modify the Target Framework and Platform Toolset	Demonstrates how to compile a project for multiple frameworks or toolsets.

See also

Use the MSVC toolset from the command line

Walkthrough: Using MSBuild to Create a Visual C++ Project

Article • 10/25/2021

This walkthrough demonstrates how to use MSBuild in a command prompt to build a Visual Studio C++ project. You'll learn how to create an XML-based `.vcxproj` project file for a Visual C++ console application. After building the project, you'll learn how to customize the build process.

> **Important**
>
> Don't use this approach if you intend to edit the project file later by using the Visual Studio IDE. If you create a `.vcxproj` file manually, the Visual Studio IDE might not be able to edit or load it, especially if the project uses wildcards in project items. For more information, see **.vcxproj and .props file structure** and **.vcxproj files and wildcards**.

This walkthrough illustrates these tasks:

- Creating the C++ source files for your project.
- Creating the XML MSBuild project file.
- Using MSBuild to build your project.
- Using MSBuild to customize your project.

Prerequisites

You need these prerequisites to complete this walkthrough:

- A copy of Visual Studio with the **Desktop development with C++** workload installed.
- A general understanding of the MSBuild system.

> **Note**
>
> Most of the low-level build instructions are contained in the `.targets` and `.props` files that are defined under the default targets folder, stored in the property

`$(VCTargetsPath)`. It's where you'll find files such as *`Microsoft.Cpp.Common.props`*. The default path for these files is under *`%VSINSTALLDIR%MSBuild\Microsoft\VC\<version>\`*. The `<version>` path element is specific to the version of Visual Studio. It's *`v160`* for Visual Studio 2019. Visual Studio 2017 stored these files under *`%VSINSTALLDIR%Common7\IDE\VC\VCTargets\`*. Visual Studio 2015 and earlier versions stored them under *`%ProgramFiles(x86)%\MSBuild\Microsoft.Cpp\v4.0\<version>\`*.

Create the C++ source files

In this walkthrough, you'll create a project that has a source file and a header file. The source file *`main.cpp`* contains the `main` function for the console application. The header file *`main.h`* contains code to include the `<iostream>` header file. You can create these C++ files by using Visual Studio or a text editor such as Visual Studio Code.

To create the C++ source files for your project

1. Create a folder for your project.

2. Create a file named *`main.cpp`* and add this code to the file:

 C++

   ```
   // main.cpp : the application source code.
   #include <iostream>
   #include "main.h"
   int main()
   {
      std::cout << "Hello, from MSBuild!\n";
      return 0;
   }
   ```

3. Create a file named *`main.h`* and add this code to the file:

 C++

   ```
   // main.h: the application header code.
   /* Additional source code to include. */
   ```

Creating the XML MSBuild Project File

An MSBuild project file is an XML file that contains a project root element (`<Project>`). In the example project you'll build, the `<Project>` element contains seven child elements:

- Three item group tags (`<ItemGroup>`) that specify project configuration and platform, source file name, and header file name.
- Three import tags (`<Import>`) that specify the location of Microsoft Visual C++ settings.
- A property group tag (`<PropertyGroup>`) that specifies project settings.

To create the MSBuild project file

1. Use a text editor to create a project file that is named *`myproject.vcxproj`*, and then add the root `<Project>` element shown here. (Use `ToolsVersion="14.0"` if you're using Visual Studio 2015, `ToolsVersion="15.0"` if you're using Visual Studio 2017, or `ToolsVersion="16.0"` if you're using Visual Studio 2019.)

 XML

   ```
   <Project DefaultTargets="Build" ToolsVersion="16.0"
   xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   </Project>
   ```

 Insert the elements in the next procedure steps between the root `<Project>` tags.

2. Add these two `<ProjectConfiguration>` child elements in an `<ItemGroup>` element. The child element specifies debug and release configurations for a 32-bit Windows operating system:

 XML

   ```
   <ItemGroup>
     <ProjectConfiguration Include="Debug|Win32">
       <Configuration>Debug</Configuration>
       <Platform>Win32</Platform>
     </ProjectConfiguration>
     <ProjectConfiguration Include="Release|Win32">
       <Configuration>Release</Configuration>
       <Platform>Win32</Platform>
     </ProjectConfiguration>
   </ItemGroup>
   ```

3. Add an `<Import>` element that specifies the path of the default C++ settings for this project:

```
XML

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.default.props" />
```

4. Add a property group element (`<PropertyGroup>`) that specifies two project properties, `<ConfigurationType>` and `<PlatformToolset>`. (Use `v140` as the `<PlatformToolset>` value if you're using Visual Studio 2015, `v141` if you're using Visual Studio 2017, or `v142` if you're using Visual Studio 2019.)

```
XML

<PropertyGroup>
  <ConfigurationType>Application</ConfigurationType>
  <PlatformToolset>v142</PlatformToolset>
</PropertyGroup>
```

5. Add an `<Import>` element that specifies the path of the current C++ settings for this project:

```
XML

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />
```

6. Add a `<ClCompile>` child element in an `<ItemGroup>` element. The child element specifies the name of the C/C++ source file to compile:

```
XML

<ItemGroup>
  <ClCompile Include="main.cpp" />
</ItemGroup>
```

> **Note**
>
> `<ClCompile>` is a *build target* and is defined in the default targets folder.

7. Add a `<ClInclude>` child element in an `<ItemGroup>` element. The child element specifies the name of the header file for the C/C++ source file:

```
```

XML

```xml
<ItemGroup>
  <ClInclude Include="main.h" />
</ItemGroup>
```

8. Add an `<Import>` element that specifies the path of the file that defines the target for this project:

XML

```xml
<Import Project="$(VCTargetsPath)\Microsoft.Cpp.Targets" />
```

Complete Project File

This code shows the complete project file that you created in the previous procedure. (Use `ToolsVersion="15.0"` for Visual Studio 2017, or `ToolsVersion="14.0"` for Visual Studio 2015.)

XML

```xml
<Project DefaultTargets="Build" ToolsVersion="16.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <ProjectConfiguration Include="Debug|Win32">
      <Configuration>Debug</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
    <ProjectConfiguration Include="Release|Win32">
      <Configuration>Release</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
  </ItemGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.default.props" />
  <PropertyGroup>
    <ConfigurationType>Application</ConfigurationType>
    <PlatformToolset>v142</PlatformToolset>
  </PropertyGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />
  <ItemGroup>
    <ClCompile Include="main.cpp" />
  </ItemGroup>
  <ItemGroup>
    <ClInclude Include="main.h" />
  </ItemGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.Targets" />
</Project>
```

Using MSBuild to Build Your Project

Enter this command at the command prompt to build your console application:

```
msbuild myproject.vcxproj /p:configuration=debug
```

MSBuild creates a folder for the output files, and then compiles and links your project to generate the *`Myproject.exe`* program. After the build process finishes, use this command to run the application from the debug folder:

```
myproject
```

The application should display "Hello, from MSBuild!" in the console window.

Customizing Your Project

MSBuild enables you to execute predefined build targets, apply user-defined properties, and use custom tools, events, and build steps. This section illustrates these tasks:

- Using MSBuild with build targets.
- Using MSBuild with build properties.
- Using MSBuild with the 64-bit compiler and tools.
- Using MSBuild with different toolsets.
- Adding MSBuild customizations.

Using MSBuild with Build Targets

A *build target* is a named set of predefined or user-defined commands that can be executed during the build. Use the target command-line option (`/t`) to specify a build target. For the `myproject` example project, the predefined `clean` target deletes all files in the debug folder and creates a new log file.

At the command prompt, enter this command to clean `myproject`:

```
msbuild myproject.vcxproj /t:clean
```

Using MSBuild with Build Properties

The property command-line option (`/p`) enables you to override a property in your project build file. In the `myproject` example project, the release or debug build configuration is specified by the `Configuration` property. The operating system that you'll use to run the built application is specified by the `Platform` property.

At the command prompt, enter this command to create a debug build of the `myproject` application to run on 32-bit Windows:

```
msbuild myproject.vcxproj /p:configuration=debug /p:platform=win32
```

Assume that the `myproject` example project also defines a configuration for 64-bit Windows, and another configuration for a custom operating system named `myplatform`.

At the command prompt, enter this command to create a release build that runs on 64-bit Windows:

```
msbuild myproject.vcxproj /p:configuration=release /p:platform=x64
```

At the command prompt, enter this command to create a release build for `myplatform`:

```
msbuild myproject.vcxproj /p:configuration=release /p:platform=myplatform
```

Using MSBuild with the 64-bit Compiler and Tools

If you have installed Visual Studio on 64-bit Windows, the 64-bit x64 native and cross tools are installed by default. You can configure MSBuild to use the 64-bit compiler and tools to build your application by setting the `PreferredToolArchitecture` property. This property doesn't affect the project configuration or platform properties. By default, the 32-bit version of the tools is used. To specify the 64-bit version of the compiler and tools, add this property group element to the *`Myproject.vcxproj`* project file after the *`Microsoft.Cpp.default.props`* file `<Import />` element:

XML

```
<PropertyGroup>
    <PreferredToolArchitecture>x64</PreferredToolArchitecture>
</PropertyGroup>
```

At the command prompt, enter this command to use the 64-bit tools to build your application:

```
msbuild myproject.vcxproj /p:PreferredToolArchitecture=x64
```

Using MSBuild with a different toolset

If you have the toolsets and libraries for other versions of Visual C++ installed, MSBuild can build applications for either the current Visual C++ version or for the other installed versions. For example, if you have installed Visual Studio 2012, to specify the Visual C++ 11.0 toolset for Windows XP, add this property group element to the *`Myproject.vcxproj`* project file after the *`Microsoft.Cpp.props`* file `<Import />` element:

XML

```
<PropertyGroup>
    <PlatformToolset>v110_xp</PlatformToolset>
</PropertyGroup>
```

To rebuild your project with the Visual C++ 11.0 Windows XP toolset, enter this command:

```
msbuild myproject.vcxproj /p:PlatformToolset=v110_xp /t:rebuild
```

Adding MSBuild customizations

MSBuild provides various ways to customize your build process. These articles show how to add custom build steps, tools, and events to your MSBuild project:

- How to: Add a Custom Build Step to MSBuild Projects
- How to: Add Custom Build Tools to MSBuild Projects
- How to: Use Build Events in MSBuild Projects

How to: Use Build Events in MSBuild Projects

Article • 08/03/2021

A build event is a command that MSBuild performs at a particular stage in the build process. The *pre-build* event occurs before the build starts; the *pre-link* event occurs before the link step starts; and the *post-build* event occurs after the build successfully ends. A build event occurs only if the associated build step occurs. For example, the pre-link event does not occur if the link step does not run.

Each of the three build events is represented in an item definition group by a command element (`<Command>`) that is executed and a message element (`<Message>`) that is displayed when **MSBuild** performs the build event. Each element is optional, and if you specify the same element multiple times, the last occurrence takes precedence.

An optional *use-in-build* element (`<`*build-event*`UseInBuild>`) can be specified in a property group to indicate whether the build event is executed. The value of the content of a *use-in-build* element is either **`true`** or **`false`**. By default, a build event is executed unless its corresponding *use-in-build* element is set to **`false`**.

The following table lists each build event XML element:

XML Element	Description
`PreBuildEvent`	This event executes before the build begins.
`PreLinkEvent`	This event executes before the link step begins.
`PostBuildEvent`	This event executes after the build finishes.

The following table lists each *use-in-build* element:

XML Element	Description
`PreBuildEventUseInBuild`	Specifies whether to execute the *pre-build* event.
`PreLinkEventUseInBuild`	Specifies whether to execute the *pre-link* event.
`PostBuildEventUseInBuild`	Specifies whether to execute the *post-build* event.

Example

The following example can be added inside of the Project element of the myproject.vcxproj file created in Walkthrough: Using MSBuild to Create a C++ Project. A *pre-build* event makes a copy of main.cpp; a *pre-link* event makes a copy of main.obj; and a *post-build* event makes a copy of myproject.exe. If the project is built using a release configuration, the build events are executed. If the project is built using a debug configuration, the build events are not executed.

XML

```
<ItemDefinitionGroup>
  <PreBuildEvent>
    <Command>copy $(ProjectDir)main.cpp
$(ProjectDir)copyOfMain.cpp</Command>
    <Message>Making a copy of main.cpp </Message>
  </PreBuildEvent>
  <PreLinkEvent>
    <Command>copy $(ProjectDir)$(Configuration)\main.obj
$(ProjectDir)$(Configuration)\copyOfMain.obj</Command>
    <Message>Making a copy of main.obj</Message>
  </PreLinkEvent>
  <PostBuildEvent>
    <Command>copy $(ProjectDir)$(Configuration)\$(TargetFileName)
$(ProjectDir)$(Configuration)\copyOfMyproject.exe</Command>
    <Message>Making a copy of myproject.exe</Message>
  </PostBuildEvent>
</ItemDefinitionGroup>

<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|Win32'">
  <PreBuildEventUseInBuild>true</PreBuildEventUseInBuild>
  <PreLinkEventUseInBuild>true</PreLinkEventUseInBuild>
  <PostBuildEventUseInBuild>true</PostBuildEventUseInBuild>
</PropertyGroup>

<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">
  <PreBuildEventUseInBuild>false</PreBuildEventUseInBuild>
  <PreLinkEventUseInBuild>false</PreLinkEventUseInBuild>
  <PostBuildEventUseInBuild>false</PostBuildEventUseInBuild>
</PropertyGroup>
```

See also

MSBuild on the command line - C++
Walkthrough: Using MSBuild to Create a C++ Project

How to: Add a Custom Build Step to MSBuild Projects

Article • 08/03/2021

A custom build step is a user-defined step in a build. A custom build step behaves like any other *command tool* step, such as the standard compile or link tool step.

Specify a custom build step in the project file (.vcxproj). The step can specify a command line to execute, any additional input or output files, and a message to display. If **MSBuild** determines that your output files are out-of-date with regard to your input files, it displays the message and executes the command.

To specify the location of the custom build step in the sequence of build targets, use one or both of the `CustomBuildAfterTargets` and `CustomBuildBeforeTargets` XML elements in the project file. For example, you could specify that the custom build step runs after the link tool target and before the manifest tool target. The actual set of available targets depends on your particular build.

Specify the `CustomBuildBeforeTargets` element to execute the custom build step before a particular target runs, the `CustomBuildAfterTargets` element to execute the step after a particular target runs, or both elements to execute the step between two adjacent targets. If neither element is specified, your custom build tool executes at its default location, which is after the **Link** target.

Custom build steps and custom build tools share the information specified in the `CustomBuildBeforeTargets` and `CustomBuildAfterTargets` XML elements. Therefore, specify those targets just one time in your project file.

To define what is executed by the custom build step

1. Add a property group to the project file. In this property group, specify the command, its inputs and outputs, and a message, as shown in the following example. This example creates a .cab file from the main.cpp file you created in Walkthrough: Using MSBuild to Create a C++ Project.

```
<ItemDefinitionGroup>
  <CustomBuildStep>
    <Command>makecab.exe $(ProjectDir)main.cpp
$(TargetName).cab</Command>
    <Outputs>$(TargetName).cab</Outputs>
```

```
    <Inputs>$(ProjectDir)main.cpp</Inputs>
  </CustomBuildStep>
</ItemDefinitionGroup>
```

To define where in the build the custom build step will execute

1. Add the following property group to the project file. You can specify both targets, or you can omit one if you just want the custom step to execute before or after a particular target. This example tells **MSBuild** to perform the custom step after the compile step but before the link step.

```

<PropertyGroup>
  <CustomBuildAfterTargets>ClCompile</CustomBuildAfterTargets>
  <CustomBuildBeforeTargets>Link</CustomBuildBeforeTargets>
</PropertyGroup>
```

See also

Walkthrough: Using MSBuild to Create a C++ Project
How to: Use Build Events in MSBuild Projects
How to: Add Custom Build Tools to MSBuild Projects

How to: Add custom build tools to MSBuild projects

Article • 03/22/2022

A custom build tool is a user-defined, command-line tool that's associated with a particular file.

For a particular file, specify in the project file (`.vcxproj`) the command line to execute, any other input or output files, and a message to display. If **MSBuild** determines that your output files are out of date relative to your input files, it displays the message and executes the command-line tool.

Specify custom build tools and custom build steps

To specify when the custom build tool executes, use one or both of the `CustomBuildBeforeTargets` and `CustomBuildAfterTargets` XML elements in the project file. For example, you might specify that your custom build tool run after the MIDL compiler and before the C/C++ compiler. Specify the `CustomBuildBeforeTargets` element to execute the tool before a particular target runs. Use the `CustomBuildAfterTargets` element to execute the tool after a particular target runs. Use both elements to run the tool between execution of two targets. If neither element is specified, your custom build tool executes at its default location, which is before the **MIDL** target.

Custom build steps and custom build tools share the information specified in the `CustomBuildBeforeTargets` and `CustomBuildAfterTargets` XML elements. Specify those targets one time in your project file.

To add a custom build tool

1. Add an item group to the project file and add an item for each input file. Specify the command and its inputs, outputs, and a message as item metadata, as shown here. This example assumes that a "faq.txt" file exists in the same directory as your project. The custom build step copies it to the output directory.

   ```
   XML
   ```

```
<ItemGroup>
  <CustomBuild Include="faq.txt">
    <Message>Copying readme...</Message>
    <Command>copy %(Identity) $(OutDir)%(Identity)</Command>
    <Outputs>$(OutDir)%(Identity)</Outputs>
  </CustomBuild>
</ItemGroup>
```

To define where in the build the custom build tools execute

1. Add the following property group to the project file. You have to specify at least one of the targets. You can omit the other if you're only interested in having your build step execute before (or after) a particular target. This example performs the custom step after compiling but before linking.

XML

```
<PropertyGroup>
  <CustomBuildAfterTargets>ClCompile</CustomBuildAfterTargets>
  <CustomBuildBeforeTargets>Link</CustomBuildBeforeTargets>
</PropertyGroup>
```

See also

Walkthrough: Using MSBuild to create a C++ project
How to: Use build events in MSBuild projects
How to: Add a custom build step to MSBuild projects
Common macros for MSBuild commands and properties
MSBuild well-known item metadata

How to: Integrate Custom Tools into the Project Properties

Article • 08/03/2021

You can add custom tool options to the Visual Studio **Property Pages** window by creating an XML file.

The **Configuration Properties** section of the **Property Pages** window displays setting groups known as *rules*. Every rule contains the settings for a tool or a group of features. For example, the **Linker** rule contains the settings for the linker tool. The settings in a rule can be subdivided into *categories*.

You can create a rule file that contains properties for your custom tool so that the properties are loaded when Visual Studio starts. For information about how to modify the file, see Platform Extensibility Part 2 on the Visual Studio Project Team blog.

The folder to place your rule file in depends on the locale and the version of Visual Studio in use. In a Visual Studio 2019 or later developer command prompt, the rules folder is `%VSINSTALLDIR%MSBuild\Microsoft\VC\<version>\<locale>\`, where the `<version>` value is `v160` in Visual Studio 2019. The `<locale>` is an LCID, for example, `1033` for English. In Visual Studio 2017, the rules folder is `%VSINSTALLDIR%Common7\IDE\VC\VCTargets\<locale>\`. In a Visual Studio 2015 or earlier developer command prompt, the rules folder is `%ProgramFiles(x86)%\MSBuild\Microsoft.Cpp\v4.0\<version>\<locale>\`. You'll use a different path for each edition of Visual Studio that's installed, and for each language. For example, the default rules folder path for Visual Studio 2019 Community edition in English could be `C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Microsoft\VC\v160\1033\`.

To add or change project properties

1. In the XML editor, create an XML file.

2. Save the file in the default rules folder. Adjust the path for your language and Visual Studio edition. Every rule in the **Property Pages** window is represented by an XML file in this folder. Make sure that the file is uniquely named in the folder.

3. Copy the content of an existing rules file, such as `rc.xml`, close it without saving changes, and then paste the content in your new XML file. You can copy any XML schema file to use as a template. Choose one that's similar to your tool.

4. In the new XML file, modify the content according to your requirements. Make sure to change the **Rule Name** and **Rule.DisplayName** at the top of the file.

5. Save your changes and close the file.

6. The XML files in the rules folder are loaded when Visual Studio starts. To test the new file, restart Visual Studio.

7. In **Solution Explorer**, right-click a project and then choose **Properties**. In the **Property Pages** window, verify that there's a new node with the name of your rule.

See also

MSBuild on the command line - C++

How to: Modify the Target Framework and Platform Toolset

Article • 11/23/2021

You can edit a Visual Studio C++ project file to target different versions of the C++ platform toolset. The Windows SDK and the .NET Framework used are also editable. (The .NET Framework applies to C++/CLI projects only). A new project uses the default .NET Framework and toolset of the Visual Studio version that you use to create the project. If you modify these values in the .vcxproj file, you can use the same code base for every compilation target.

Platform toolset

The platform toolset consists of the C++ compiler (cl.exe) and linker (link.exe), along with the C/C++ standard libraries. Visual Studio 2015, Visual Studio 2017, and Visual Studio 2019 are binary-compatible. It's shown by the major version of the toolset, which has remained at 14. Projects compiled in Visual Studio 2019 or Visual Studio 2017 are ABI-backwards-compatible with 2017 and 2015 projects. The minor version has updated by 1 for each version since Visual Studio 2015:

- Visual Studio 2015: v140
- Visual Studio 2017: v141
- Visual Studio 2019: v142
- Visual Studio 2022: v143

These toolsets support the .NET Framework 4.5 and later.

Visual Studio also supports multitargeting for C++ projects. You can use the latest Visual Studio IDE to edit and build projects created by older versions of Visual Studio. It doesn't require a project upgrade the projects to use a new version of the toolset. It does require the older toolset is installed on your computer. For more information, see How to use native multi-targeting in Visual Studio. For example, in Visual Studio 2015, you can *target* .NET Framework 2.0, but you must use an earlier toolset that supports it.

Target framework (C++/CLI project only)

When you change the target Framework, also change the platform toolset to a version that supports that Framework. For example, to target the .NET Framework 4.5, you must use a compatible platform toolset. These toolsets include Visual Studio 2015 (v140),

Visual Studio 2013 (v120), or Visual Studio 2012 (v110). You can use the Windows 7.1 SDK to target .NET Framework 2.0, 3.0, 3.5, and 4.

You can extend the target platform further by creating a custom platform toolset. For more information, see C++ Native Multi-Targeting on the Visual C++ blog.

To change the target Framework

1. In Visual Studio, in **Solution Explorer**, select your project. On the menu bar, open the **Project** menu and choose **Unload project**. This command unloads the project (.vcxproj) file for your project.

 > **Note**
 >
 > A C++ project can't be loaded while you edit the project file in Visual Studio. However, you can use another editor such as Notepad to modify the project file while the project is loaded in Visual Studio. Visual Studio will detect that the project file has changed and prompt you to reload the project.

2. On the menu bar, select **File**, **Open**, **File**. In the **Open File** dialog box, navigate to your project folder, and then open the project (.vcxproj) file.

3. In the project file, locate the entry for the target Framework version. For example, if your project is designed to use the .NET Framework 4.5, locate `<TargetFrameworkVersion>v4.5</TargetFrameworkVersion>` in the `<PropertyGroup Label="Globals">` element of the `<Project>` element. If the `<TargetFrameworkVersion>` element isn't present, your project doesn't use the .NET Framework and no change is required.

4. Change the value to the Framework version you want, such as v3.5 or v4.6.

5. Save the changes and close the editor.

6. In **Solution Explorer**, open the shortcut menu for your project and then choose **Reload Project**.

7. To verify the change, on the menu bar, select **Project** > **Properties** to open your project **Property Pages** dialog box. In the dialog box, select the **Configuration Properties** > **General** property page. Verify that **.NET Target Framework Version** shows the new Framework version.

To change the platform toolset

1. In Visual Studio, on the menu bar, select **Project** > **Properties** to open your project **Property Pages** dialog box.
2. In the top of the **Property Pages** dialog box, open the **Configuration** drop-down list and then select **All Configurations**.
3. In the dialog box, select the **Configuration Properties** > **General** property page.
4. In the properties page, select **Platform Toolset** and then select the toolset you want from the drop-down list. For example, if you've installed the Visual Studio 2010 toolset, select **Visual Studio 2010 (v100)** to use it for your project.
5. Choose the **OK** button to save your changes.

Next Steps

Walkthrough: Working with Projects and Solutions (C++)

See also

MSBuild on the command line - C++

Walkthrough: Create and use a static library

Article • 10/29/2021

This step-by-step walkthrough shows how to create a static library (.lib file) for use with C++ apps. Using a static library is a great way to reuse code. Rather than reimplementing the same routines in every app that requires the functionality, you write them one time in a static library and then reference it from the apps. Code linked from a static library becomes part of your app—you don't have to install another file to use the code.

This walkthrough covers these tasks:

- Create a static library project
- Add a class to the static library
- Create a C++ console app that references the static library
- Use the functionality from the static library in the app
- Run the app

Prerequisites

An understanding of the fundamentals of the C++ language.

Create a static library project

The instructions for how to create the project vary depending on your version of Visual Studio. To see the documentation for your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

To create a static library project in Visual Studio

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a New Project** dialog.
2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Library**.

3. From the filtered list of project types, select **Windows Desktop Wizard**, then choose **Next**.

4. In the **Configure your new project** page, enter *MathLibrary* in the **Project name** box to specify a name for the project. Enter *StaticMath* in the **Solution name** box. Choose the **Create** button to open the **Windows Desktop Project** dialog.

5. In the **Windows Desktop Project** dialog, under **Application type**, select **Static Library (.lib)**.

6. Under **Additional options**, uncheck the **Precompiled header** check box if it's checked. Check the **Empty project** box.

7. Choose **OK** to create the project.

Add a class to the static library

To add a class to the static library

1. To create a header file for a new class, right-click to open the shortcut menu for the **MathLibrary** project in **Solution Explorer**, and then choose **Add** > **New Item**.

2. In the **Add New Item** dialog box, select **Visual C++** > **Code**. In the center pane, select **Header File (.h)**. Specify a name for the header file—for example, *MathLibrary.h*—and then choose the **Add** button. A nearly blank header file is displayed.

3. Add a declaration for a class named `Arithmetic` to do common mathematical operations such as addition, subtraction, multiplication, and division. The code should resemble:

```
C++
```

```
// MathLibrary.h
#pragma once

namespace MathLibrary
{
    class Arithmetic
    {
    public:
        // Returns a + b
        static double Add(double a, double b);

        // Returns a - b
        static double Subtract(double a, double b);
```

```
        // Returns a * b
        static double Multiply(double a, double b);

        // Returns a / b
        static double Divide(double a, double b);
    };
}
```

4. To create a source file for the new class, open the shortcut menu for the **MathLibrary** project in **Solution Explorer**, and then choose **Add** > **New Item**.

5. In the **Add New Item** dialog box, in the center pane, select **C++ File (.cpp)**. Specify a name for the source file—for example, *MathLibrary.cpp*—and then choose the **Add** button. A blank source file is displayed.

6. Use this source file to implement the functionality for class `Arithmetic`. The code should resemble:

C++

```
// MathLibrary.cpp
// compile with: cl /c /EHsc MathLibrary.cpp
// post-build command: lib MathLibrary.obj

#include "MathLibrary.h"

namespace MathLibrary
{
    double Arithmetic::Add(double a, double b)
    {
        return a + b;
    }

    double Arithmetic::Subtract(double a, double b)
    {
        return a - b;
    }

    double Arithmetic::Multiply(double a, double b)
    {
        return a * b;
    }

    double Arithmetic::Divide(double a, double b)
    {
        return a / b;
    }
}
```

7. To build the static library, select **Build** > **Build Solution** on the menu bar. The build creates a static library, *MathLibrary.lib*, that can be used by other programs.

> **Note**
>
> When you build on the Visual Studio command line, you must build the program in two steps. First, run `cl /c /EHsc MathLibrary.cpp` to compile the code and create an object file that's named *MathLibrary.obj*. (The `cl` command invokes the compiler, Cl.exe, and the `/c` option specifies compile without linking. For more information, see **/c (Compile Without Linking)**.) Second, run `lib MathLibrary.obj` to link the code and create the static library *MathLibrary.lib*. (The `lib` command invokes the Library Manager, Lib.exe. For more information, see **LIB Reference**.)

Create a C++ console app that references the static library

To create a C++ console app that references the static library in Visual Studio

1. In **Solution Explorer**, right-click on the top node, **Solution 'StaticMath'**, to open the shortcut menu. Choose **Add** > **New Project** to open the **Add a New Project** dialog.

2. At the top of the dialog, set the **Project type** filter to **Console**.

3. From the filtered list of project types, choose **Console App** then choose **Next**. In the next page, enter *MathClient* in the **Name** box to specify a name for the project.

4. Choose the **Create** button to create the client project.

5. After you create a console app, an empty program is created for you. The name for the source file is the same as the name that you chose earlier. In the example, it's named `MathClient.cpp`.

Use the functionality from the static library in the app

To use the functionality from the static library in the app

1. Before you can use the math routines in the static library, you must reference it. Open the shortcut menu for the **MathClient** project in **Solution Explorer**, and then choose **Add** > **Reference**.

2. The **Add Reference** dialog box lists the libraries that you can reference. The **Projects** tab lists the projects in the current solution and any libraries they reference. Open the **Projects** tab, select the **MathLibrary** check box, and then choose the **OK** button.

3. To reference the `MathLibrary.h` header file, you must modify the included directories path. In **Solution Explorer**, right-click on **MathClient** to open the shortcut menu. Choose **Properties** to open the **MathClient Property Pages** dialog box.

4. In the **MathClient Property Pages** dialog box, set the **Configuration** drop-down to **All Configurations**. Set the **Platform** drop-down to **All Platforms**.

5. Select the **Configuration Properties** > **C/C++** > **General** property page. In the **Additional Include Directories** property, specify the path of the **MathLibrary** directory, or browse for it.

 To browse for the directory path:

 a. Open the **Additional Include Directories** property value drop-down list, and then choose **Edit**.

 b. In the **Additional Include Directories** dialog box, double-click in the top of the text box. Then choose the ellipsis button (**...**) at the end of the line.

 c. In the **Select Directory** dialog box, navigate up a level, and then select the **MathLibrary** directory. Then choose the **Select Folder** button to save your selection.

 d. In the **Additional Include Directories** dialog box, choose the **OK** button.

 e. In the **Property Pages** dialog box, choose the **OK** button to save your changes to the project.

6. You can now use the `Arithmetic` class in this app by including the `#include "MathLibrary.h"` header in your code. Replace the contents of `MathClient.cpp` with this code:

   ```
   C++
   ```

```
// MathClient.cpp
// compile with: cl /EHsc MathClient.cpp /link MathLibrary.lib

#include <iostream>
#include "MathLibrary.h"

int main()
{
    double a = 7.4;
    int b = 99;

    std::cout << "a + b = " <<
        MathLibrary::Arithmetic::Add(a, b) << std::endl;
    std::cout << "a - b = " <<
        MathLibrary::Arithmetic::Subtract(a, b) << std::endl;
    std::cout << "a * b = " <<
        MathLibrary::Arithmetic::Multiply(a, b) << std::endl;
    std::cout << "a / b = " <<
        MathLibrary::Arithmetic::Divide(a, b) << std::endl;

    return 0;
}
```

7. To build the executable, choose **Build** > **Build Solution** on the menu bar.

Run the app

To run the app

1. Make sure that **MathClient** is selected as the default project. To select it, right-click to open the shortcut menu for **MathClient** in **Solution Explorer**, and then choose **Set as StartUp Project**.

2. To run the project, on the menu bar, choose **Debug** > **Start Without Debugging**. The output should resemble:

Output

```
a + b = 106.4
a - b = -91.6
a * b = 732.6
a / b = 0.0747475
```

See also

Walkthrough: Creating and Using a Dynamic Link Library (C++)

Desktop Applications (Visual C++)

Create C/C++ DLLs in Visual Studio

Article • 08/03/2021

In Windows, a dynamic-link library (DLL) is a kind of executable file that acts as a shared library of functions and resources. Dynamic linking is an operating system capability. It enables an executable to call functions or use resources stored in a separate file. These functions and resources can be compiled and deployed separately from the executables that use them.

A DLL isn't a stand-alone executable. DLLs run in the context of the applications that call them. The operating system loads the DLL into an application's memory space. It's done either when the application is loaded (*implicit linking*), or on demand at runtime (*explicit linking*). DLLs also make it easy to share functions and resources across executables. Multiple applications can access the contents of a single copy of a DLL in memory at the same time.

Differences between dynamic linking and static linking

Static linking copies all the object code in a static library into the executables that use it when they're built. Dynamic linking includes only the information needed by Windows at run time to locate and load the DLL that contains a data item or function. When you create a DLL, you also create an import library that contains this information. When you build an executable that calls the DLL, the linker uses the exported symbols in the import library to store this information for the Windows loader. When the loader loads a DLL, the DLL is mapped into the memory space of your application. If present, a special function in the DLL, `DllMain`, is called to do any initialization the DLL requires.

Differences between applications and DLLs

Even though DLLs and applications are both executable modules, they differ in several ways. The most obvious difference is that you can't run a DLL. From the system's point of view, there are two fundamental differences between applications and DLLs:

- An application can have multiple instances of itself running in the system simultaneously. A DLL can have only one instance.
- An application can be loaded as a process. It can own things such as a stack, threads of execution, global memory, file handles, and a message queue. A DLL

can't own these things.

Advantages of using DLLs

Dynamic linking to code and resources offers several advantages over static linking:

- Dynamic linking saves memory and reduces swapping. Many processes can use a DLL simultaneously, sharing a single copy of the read-only parts of a DLL in memory. In contrast, every application that is built by using a statically linked library has a complete copy of the library code that Windows must load into memory.

- Dynamic linking saves disk space and bandwidth. Many applications can share a single copy of the DLL on disk. In contrast, each application built by using a static link library has the library code linked into its executable image. That uses more disk space, and takes more bandwidth to transfer.

- Maintenance, security fixes, and upgrades can be easier. When your applications use common functions in a DLL, you can implement bug fixes and deploy updates to the DLL. When DLLs are updated, the applications that use them don't need to be recompiled or relinked. They can make use of the new DLL as soon as it's deployed. In contrast, when you make fixes in statically linked object code, you must relink and redeploy every application that uses it.

- You can use DLLs to provide after-market support. For example, a display driver DLL can be modified to support a display that wasn't available when the application was shipped.

- You can use explicit linking to discover and load DLLs at runtime. For example, application extensions that add new functionality to your app without rebuilding or redeploying it.

- Dynamic linking makes it easier to support applications written in different programming languages. Programs written in different programming languages can call the same DLL function as long as the programs follow the function's calling convention. The programs and the DLL function must be compatible in the following ways: The order in which the function expects its arguments to be pushed onto the stack. Whether the function or the application is responsible for cleaning up the stack. And, whether any arguments are passed in registers.

- Dynamic linking provides a mechanism to extend the Microsoft Foundation Class library (MFC) classes. You can derive classes from the existing MFC classes and place them in an MFC extension DLL for use by MFC applications.

- Dynamic linking makes creation of international versions of your application easier. DLLs are a convenient way to supply locale-specific resources, which make it much easier to create international versions of an application. Instead of shipping many localized versions of your application, you can place the strings and images for each language in a separate resource DLL. Then your application can load the appropriate resources for that locale at runtime.

A potential disadvantage to using DLLs is that the application isn't self-contained. It depends on the existence of a separate DLL module: one that you must deploy or verify yourself as part of your installation.

More information on how to create and use DLLs

The following articles provide detailed information about how to create C/C++ DLLs in Visual Studio.

Walkthrough: Creating and using a dynamic link library (C++)
Describes how to create and use a DLL using Visual Studio.

Kinds of DLLs
Provides information about the different kinds of DLLs that can be built.

DLL frequently asked questions
Provides answers to frequently asked questions about DLLs.

Link an executable to a DLL
Describes explicit and implicit linking to a DLL.

Initialize a DLL
Discusses DLL initialization code that must execute when your DLL loads.

DLLs and Visual C++ run-time library behavior
Describes the run-time library DLL startup sequence.

LoadLibrary and AfxLoadLibrary
Discusses using `LoadLibrary` and `AfxLoadLibrary` to explicitly link to a DLL at runtime.

GetProcAddress
Discusses using `GetProcAddress` to obtain the address of an exported function in the DLL.

FreeLibrary and AfxFreeLibrary
Discusses using `FreeLibrary` and `AfxFreeLibrary` when the DLL module is no longer needed.

Dynamic-Link Library Search Order
Describes the search path that the Windows operating system uses to locate a DLL on the system.

Module states of a regular MFC DLL dynamically linked to MFC
Describes the module states of a regular MFC DLL dynamically linked to MFC.

MFC extension DLLs
Explains DLLs that typically implement reusable classes derived from the existing MFC classes.

Creating a resource-only DLL
Discusses a resource-only DLL, which contains nothing but resources, such as icons, bitmaps, strings, and dialog boxes.

Localized resources in MFC Applications: Satellite DLLs
Provides enhanced support for satellite DLLs, a feature that helps in creating applications localized for multiple languages.

Importing and exporting
Describes importing public symbols into an application or exporting functions from a DLL

Active technology and DLLs
Allows object servers to be implemented inside a DLL.

Automation in a DLL
Describes what the Automation option in the MFC DLL Wizard supplies.

Naming conventions for MFC DLLs
Discusses how the DLLs and libraries included in MFC follow a structured naming convention.

Calling DLL functions from Visual Basic applications
Describes how to call DLL functions from Visual Basic applications.

Related Sections

Using MFC as part of a DLL
Describes regular MFC DLLs, which let you use the MFC library as part of a Windows

dynamic-link library.

DLL version of MFC

Describes how you can use the MFCxx.dll and MFCxxD.dll (where x is the MFC version number) shared dynamic-link libraries with MFC applications and MFC extension DLLs.

Walkthrough: Create and use your own Dynamic Link Library (C++)

Article • 12/10/2021

This step-by-step walkthrough shows how to use the Visual Studio IDE to create your own dynamic link library (DLL) written in Microsoft C++ (MSVC). Then it shows how to use the DLL from another C++ app. DLLs (also known as *shared libraries* in UNIX-based operating systems) are one of the most useful kinds of Windows components. You can use them as a way to share code and resources, and to shrink the size of your apps. DLLs can even make it easier to service and extend your apps.

In this walkthrough, you'll create a DLL that implements some math functions. Then you'll create a console app that uses the functions from the DLL. You'll also get an introduction to some of the programming techniques and conventions used in Windows DLLs.

This walkthrough covers these tasks:

- Create a DLL project in Visual Studio.
- Add exported functions and variables to the DLL.
- Create a console app project in Visual Studio.
- Use the functions and variables imported from the DLL in the console app.
- Run the completed app.

Like a statically linked library, a DLL *exports* variables, functions, and resources by name. A client app *imports* the names to use those variables, functions, and resources. Unlike a statically linked library, Windows connects the imports in your app to the exports in a DLL at load time or at run time, instead of connecting them at link time. Windows requires extra information that isn't part of the standard C++ compilation model to make these connections. The MSVC compiler implements some Microsoft-specific extensions to C++ to provide this extra information. We explain these extensions as we go.

This walkthrough creates two Visual Studio solutions; one that builds the DLL, and one that builds the client app. The DLL uses the C calling convention. It can be called from apps written in other programming languages, as long as the platform, calling conventions, and linking conventions match. The client app uses *implicit linking*, where

Windows links the app to the DLL at load-time. This linking lets the app call the DLL-supplied functions just like the functions in a statically linked library.

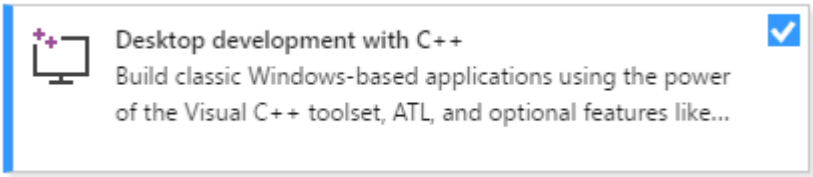
This walkthrough doesn't cover some common situations. The code doesn't show the use of C++ DLLs by other programming languages. It doesn't show how to create a resource-only DLL, or how to use explicit linking to load DLLs at run-time rather than at load-time. Rest assured, you can use MSVC and Visual Studio to do all these things.

Even though the code of the DLL is written in C++, we've used C-style interfaces for the exported functions. There are two main reasons for this: First, many other languages support imports of C-style functions. The client app doesn't have to be written in C++. Second, it avoids some common pitfalls related to exported classes and member functions. It's easy to make hard-to-diagnose errors when exporting classes, since everything referred to within a class declaration has to have an instantiation that's also exported. This restriction applies to DLLs, but not static libraries. If your classes are plain-old-data style, you shouldn't run into this issue.

For links to more information about DLLs, see Create C/C++ DLLs in Visual Studio. For more information about implicit linking and explicit linking, see Determine which linking method to use. For information about creating C++ DLLs for use with programming languages that use C-language linkage conventions, see Exporting C++ functions for use in C-language executables. For information about how to create DLLs for use with .NET languages, see Calling DLL Functions from Visual Basic Applications.

Prerequisites

- A computer that runs Microsoft Windows 7 or later versions. We recommend the latest version of Windows for the best development experience.
- A copy of Visual Studio. For information on how to download and install Visual Studio, see Install Visual Studio. When you run the installer, make sure that the **Desktop development with C++** workload is checked. Don't worry if you didn't install this workload when you installed Visual Studio. You can run the installer again and install it now.



- An understanding of the basics of using the Visual Studio IDE. If you've used Windows desktop apps before, you can probably keep up. For an introduction, see

Visual Studio IDE feature tour.

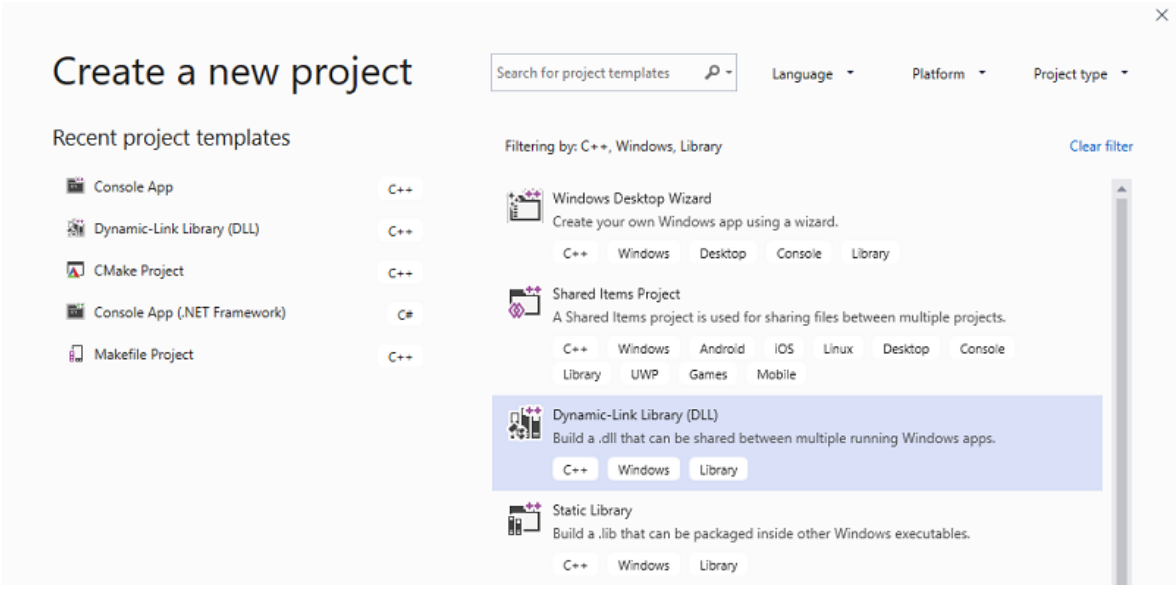
- An understanding of enough of the fundamentals of the C++ language to follow along. Don't worry, we don't do anything too complicated.

Create the DLL project

In this set of tasks, you create a project for your DLL, add code, and build it. To begin, start the Visual Studio IDE, and sign in if you need to. The instructions vary slightly depending on which version of Visual Studio you're using. Make sure you have the correct version selected in the control in the upper left of this page.

To create a DLL project in Visual Studio 2019

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a New Project** dialog box.



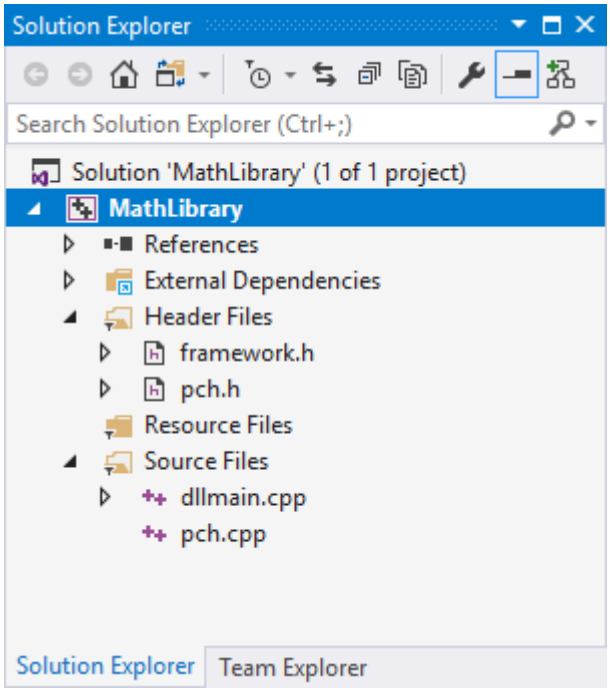
2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Library**.

3. From the filtered list of project types, select **Dynamic-link Library (DLL)**, and then choose **Next**.

4. In the **Configure your new project** page, enter *MathLibrary* in the **Project name** box to specify a name for the project. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Uncheck **Place solution and project in the same directory** if it's checked.

5. Choose the **Create** button to create the project.

When the solution is created, you can see the generated project and source files in the **Solution Explorer** window in Visual Studio.

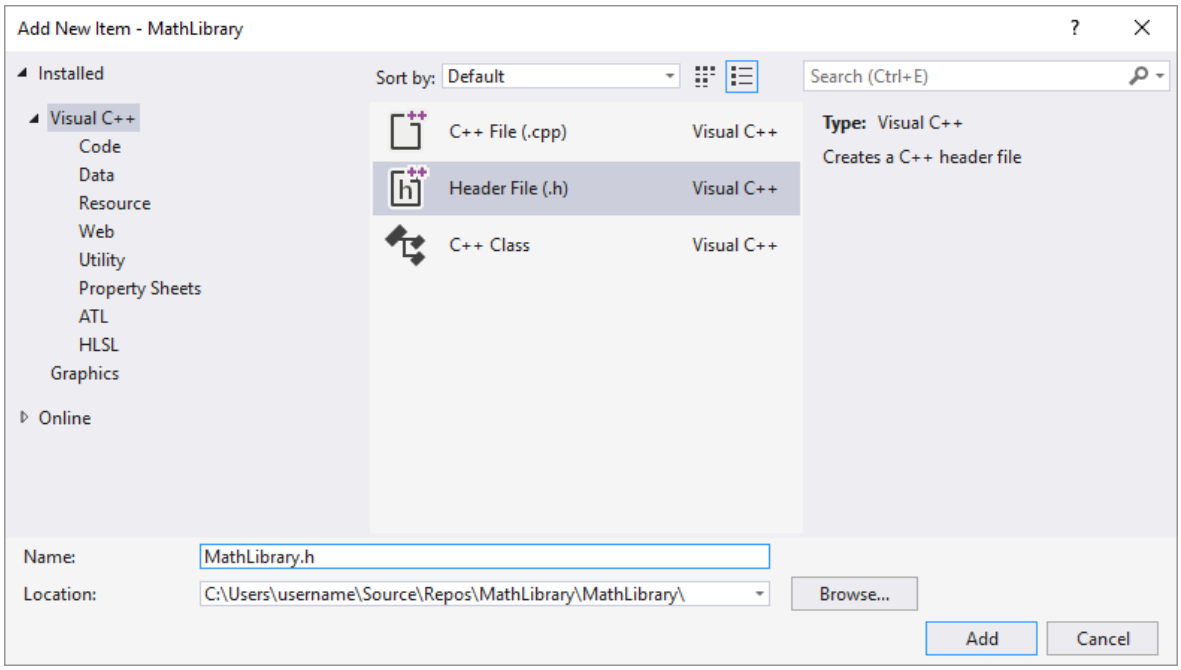


Right now, this DLL doesn't do very much. Next, you'll create a header file to declare the functions your DLL exports, and then add the function definitions to the DLL to make it more useful.

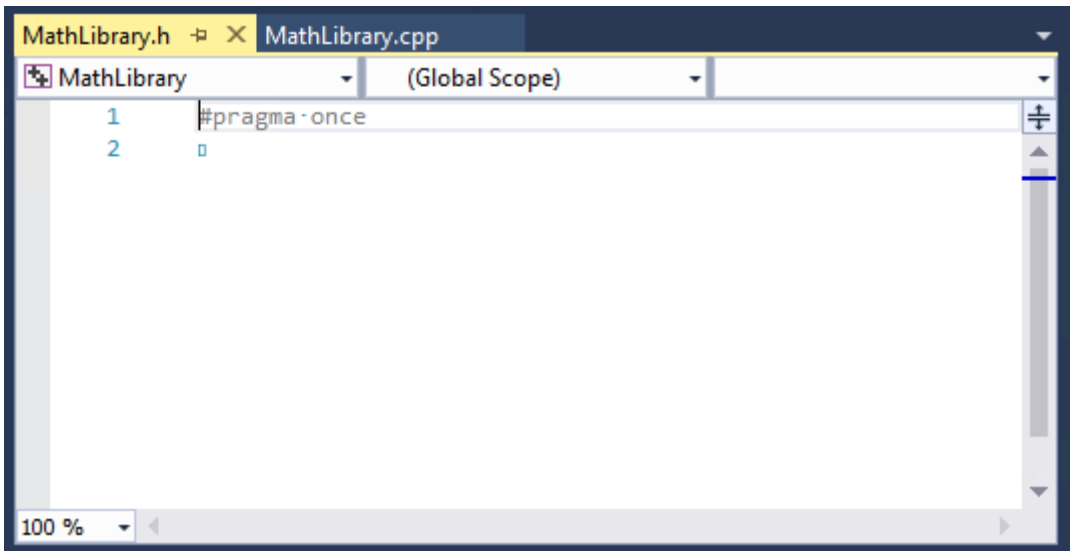
To add a header file to the DLL

1. To create a header file for your functions, on the menu bar, choose **Project** > **Add New Item**.

2. In the **Add New Item** dialog box, in the left pane, select **Visual C++**. In the center pane, select **Header File (.h)**. Specify *MathLibrary.h* as the name for the header file.



3. Choose the **Add** button to generate a blank header file, which is displayed in a new editor window.



4. Replace the contents of the header file with this code:

C++

```
// MathLibrary.h - Contains declarations of math functions
#pragma once

#ifdef MATHLIBRARY_EXPORTS
#define MATHLIBRARY_API __declspec(dllexport)
#else
#define MATHLIBRARY_API __declspec(dllimport)
#endif

// The Fibonacci recurrence relation describes a sequence F
// where F(n) is { n = 0, a
//               { n = 1, b
```

```
//               { n > 1, F(n-2) + F(n-1)
// for some initial integral values a and b.
// If the sequence is initialized F(0) = 1, F(1) = 1,
// then this relation produces the well-known Fibonacci
// sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

// Initialize a Fibonacci relation sequence
// such that F(0) = a, F(1) = b.
// This function must be called before any other function.
extern "C" MATHLIBRARY_API void fibonacci_init(
    const unsigned long long a, const unsigned long long b);

// Produce the next value in the sequence.
// Returns true on success and updates current value and index;
// false on overflow, leaves current value and index unchanged.
extern "C" MATHLIBRARY_API bool fibonacci_next();

// Get the current value in the sequence.
extern "C" MATHLIBRARY_API unsigned long long fibonacci_current();

// Get the position of the current value in the sequence.
extern "C" MATHLIBRARY_API unsigned fibonacci_index();
```

This header file declares some functions to produce a generalized Fibonacci sequence, given two initial values. A call to `fibonacci_init(1, 1)` generates the familiar Fibonacci number sequence.

Notice the preprocessor statements at the top of the file. The new project template for a DLL project adds `<PROJECTNAME>_EXPORTS` to the defined preprocessor macros. In this example, Visual Studio defines `MATHLIBRARY_EXPORTS` when your MathLibrary DLL project is built.

When the `MATHLIBRARY_EXPORTS` macro is defined, the `MATHLIBRARY_API` macro sets the `__declspec(dllexport)` modifier on the function declarations. This modifier tells the compiler and linker to export a function or variable from the DLL for use by other applications. When `MATHLIBRARY_EXPORTS` is undefined, for example, when the header file is included by a client application, `MATHLIBRARY_API` applies the `__declspec(dllimport)` modifier to the declarations. This modifier optimizes the import of the function or variable in an application. For more information, see dllexport, dllimport.

To add an implementation to the DLL

1. In **Solution Explorer**, right-click on the **Source Files** node and choose **Add** > **New Item**. Create a new .cpp file called *MathLibrary.cpp*, in the same way that you added a new header file in the previous step.

2. In the editor window, select the tab for **MathLibrary.cpp** if it's already open. If not, in **Solution Explorer**, double-click **MathLibrary.cpp** in the **Source Files** folder of the **MathLibrary** project to open it.

3. In the editor, replace the contents of the MathLibrary.cpp file with the following code:

C++

```
// MathLibrary.cpp : Defines the exported functions for the DLL.
#include "pch.h" // use stdafx.h in Visual Studio 2017 and earlier
#include <utility>
#include <limits.h>
#include "MathLibrary.h"

// DLL internal state variables:
static unsigned long long previous_;  // Previous value, if any
static unsigned long long current_;   // Current sequence value
static unsigned index_;               // Current seq. position

// Initialize a Fibonacci relation sequence
// such that F(0) = a, F(1) = b.
// This function must be called before any other function.
void fibonacci_init(
    const unsigned long long a,
    const unsigned long long b)
{
    index_ = 0;
    current_ = a;
    previous_ = b; // see special case when initialized
}

// Produce the next value in the sequence.
// Returns true on success, false on overflow.
bool fibonacci_next()
{
    // check to see if we'd overflow result or position
    if ((ULLONG_MAX - previous_ < current_) ||
        (UINT_MAX == index_))
    {
        return false;
    }

    // Special case when index == 0, just return b value
    if (index_ > 0)
    {
        // otherwise, calculate next sequence value
        previous_ += current_;
    }
    std::swap(current_, previous_);
    ++index_;
    return true;
}
```

```
// Get the current value in the sequence.
unsigned long long fibonacci_current()
{
    return current_;
}

// Get the current index position in the sequence.
unsigned fibonacci_index()
{
    return index_;
}
```

To verify that everything works so far, compile the dynamic link library. To compile, choose **Build** > **Build Solution** on the menu bar. The DLL and related compiler output are placed in a folder called *Debug* directly below the solution folder. If you create a Release build, the output is placed in a folder called *Release*. The output should look something like this:

Output

```
1>------ Build started: Project: MathLibrary, Configuration: Debug Win32 ---
---
1>pch.cpp
1>dllmain.cpp
1>MathLibrary.cpp
1>Generating Code...
1>   Creating library
C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.lib and object
C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.exp
1>MathLibrary.vcxproj ->
C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Congratulations, you've created a DLL using Visual Studio! Next, you'll create a client app that uses the functions exported by the DLL.

Create a client app that uses the DLL

When you create a DLL, think about how client apps may use it. To call the functions or access the data exported by a DLL, client source code must have the declarations available at compile time. At link time, the linker requires information to resolve the function calls or data accesses. A DLL supplies this information in an *import library*, a file that contains information about how to find the functions and data, instead of the actual code. And at run time, the DLL must be available to the client, in a location that the operating system can find.

Whether it's your own or from a third-party, your client app project needs several pieces of information to use a DLL. It needs to find the headers that declare the DLL exports, the import libraries for the linker, and the DLL itself. One solution is to copy all of these files into your client project. For third-party DLLs that are unlikely to change while your client is in development, this method may be the best way to use them. However, when you also build the DLL, it's better to avoid duplication. If you make a local copy of DLL files that are under development, you may accidentally change a header file in one copy but not the other, or use an out-of-date library.

To avoid out-of-sync code, we recommend you set the include path in your client project to include the DLL header files directly from your DLL project. Also, set the library path in your client project to include the DLL import libraries from the DLL project. And finally, copy the built DLL from the DLL project into your client build output directory. This step allows your client app to use the same DLL code you build.

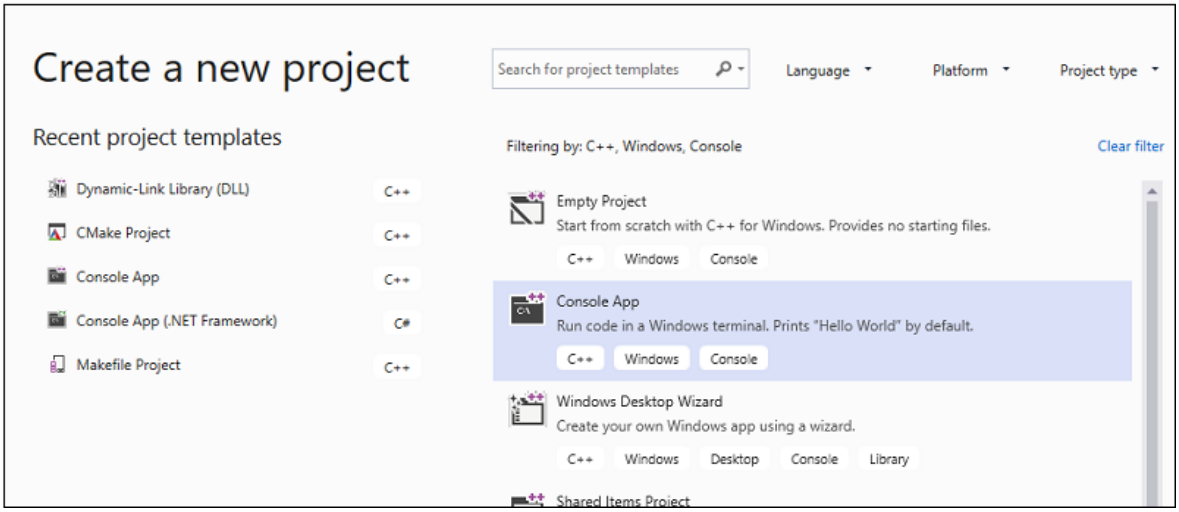
To create a client app in Visual Studio

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a new project** dialog box.

2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Console**.

3. From the filtered list of project types, choose **Console App** then choose **Next**.

4. In the **Configure your new project** page, enter *MathClient* in the **Project name** box to specify a name for the project. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Uncheck **Place solution and project in the same directory** if it's checked.



5. Choose the **Create** button to create the client project.

A minimal console application project is created for you. The name for the main source file is the same as the project name that you entered earlier. In this example, it's named **MathClient.cpp**. You can build it, but it doesn't use your DLL yet.

Next, to call the MathLibrary functions in your source code, your project must include the *MathLibrary.h* file. You could copy this header file into your client app project, then add it to the project as an existing item. This method can be a good choice for third-party libraries. However, if you're working on the code for your DLL and your client at the same time, the header files could get out of sync. To avoid this issue, set the **Additional Include Directories** path in your project to include the path to the original header.

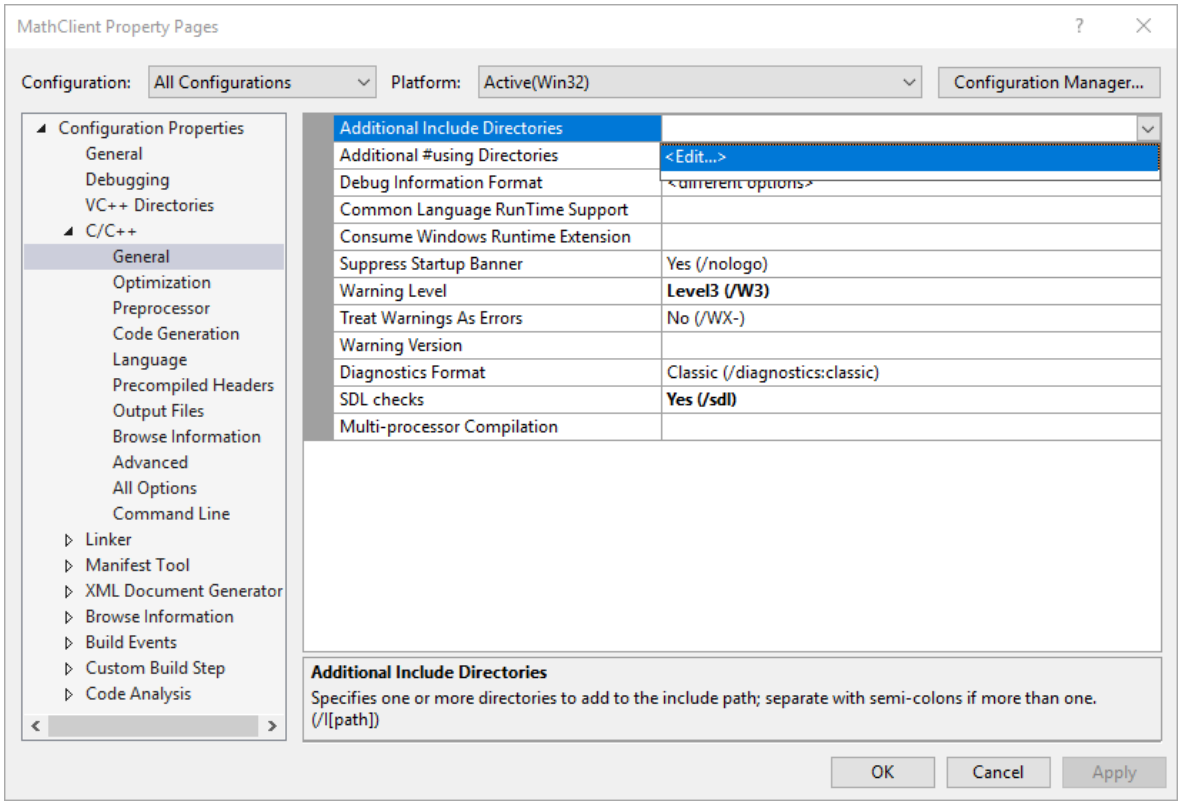
To add the DLL header to your include path

1. Right-click on the **MathClient** node in **Solution Explorer** to open the **Property Pages** dialog.

2. In the **Configuration** drop-down box, select **All Configurations** if it's not already selected.

3. In the left pane, select **Configuration Properties** > **C/C++** > **General**.

4. In the property pane, select the drop-down control next to the **Additional Include Directories** edit box, and then choose **Edit**.



5. Double-click in the top pane of the **Additional Include Directories** dialog box to enable an edit control. Or, choose the folder icon to create a new entry.

6. In the edit control, specify the path to the location of the **MathLibrary.h** header file. You can choose the ellipsis (...) control to browse to the correct folder.

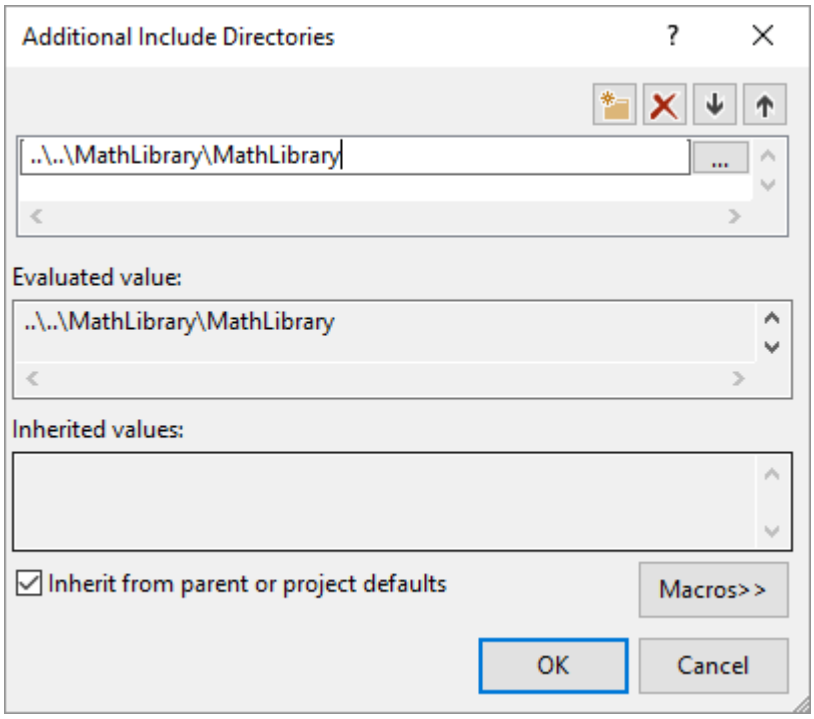
 You can also enter a relative path from your client source files to the folder that contains the DLL header files. If you followed the directions to put your client project in a separate solution from the DLL, the relative path should look like this:

   ```
   ..\..\MathLibrary\MathLibrary
   ```

 If your DLL and client projects are in the same solution, the relative path might look like this:

   ```
   ..\MathLibrary
   ```

 When the DLL and client projects are in other folders, adjust the relative path to match. Or, use the ellipsis control to browse for the folder.



7. After you've entered the path to the header file in the **Additional Include Directories** dialog box, choose the **OK** button. In the **Property Pages** dialog box, choose the **OK** button to save your changes.

You can now include the **MathLibrary.h** file and use the functions it declares in your client application. Replace the contents of **MathClient.cpp** by using this code:

```
C++
```

```
// MathClient.cpp : Client app for MathLibrary DLL.
// #include "pch.h" Uncomment for Visual Studio 2017 and earlier
#include <iostream>
#include "MathLibrary.h"

int main()
{
    // Initialize a Fibonacci relation sequence.
    fibonacci_init(1, 1);
    // Write out the sequence values until overflow.
    do {
        std::cout << fibonacci_index() << ": "
            << fibonacci_current() << std::endl;
    } while (fibonacci_next());
    // Report count of values written before overflow.
    std::cout << fibonacci_index() + 1 <<
        " Fibonacci sequence values fit in an " <<
        "unsigned 64-bit integer." << std::endl;
}
```

This code can be compiled, but not linked. If you build the client app now, the error list shows several LNK2019 errors. That's because your project is missing some information: You haven't specified that your project has a dependency on the *MathLibrary.lib* library yet. And, you haven't told the linker how to find the *MathLibrary.lib* file.

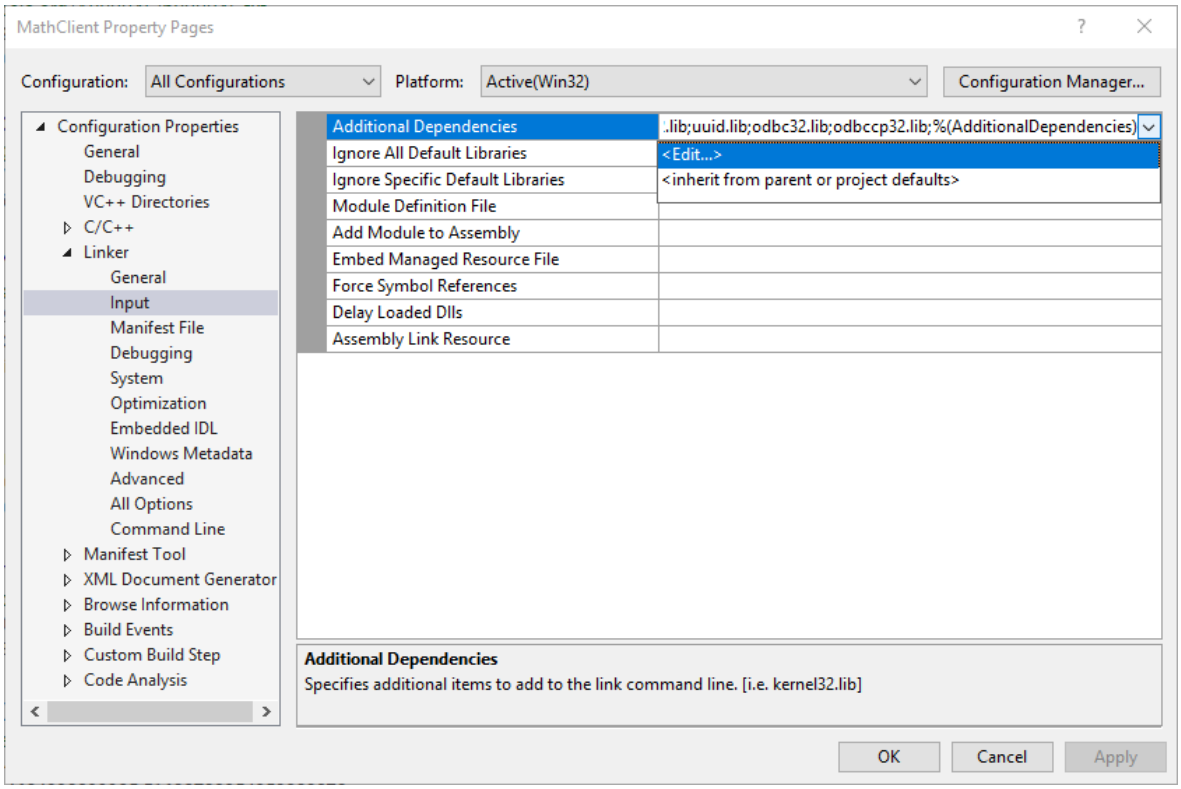
To fix this issue, you could copy the library file directly into your client app project. The linker would find and use it automatically. However, if both the library and the client app are under development, that might lead to changes in one copy that aren't shown in the other. To avoid this issue, you can set the **Additional Dependencies** property to tell the build system that your project depends on *MathLibrary.lib*. And, you can set an **Additional Library Directories** path in your project to include the path to the original library when you link.

To add the DLL import library to your project

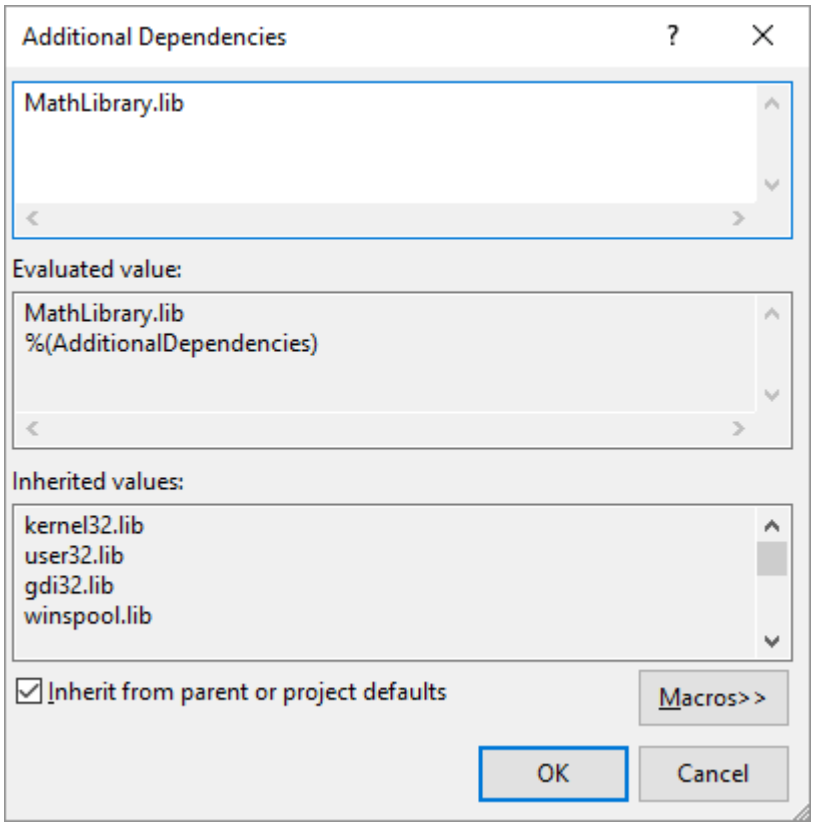
1. Right-click on the **MathClient** node in **Solution Explorer** and choose **Properties** to open the **Property Pages** dialog.

2. In the **Configuration** drop-down box, select **All Configurations** if it's not already selected. It ensures that any property changes apply to both Debug and Release builds.

3. In the left pane, select **Configuration Properties** > **Linker** > **Input**. In the property pane, select the drop-down control next to the **Additional Dependencies** edit box, and then choose **Edit**.

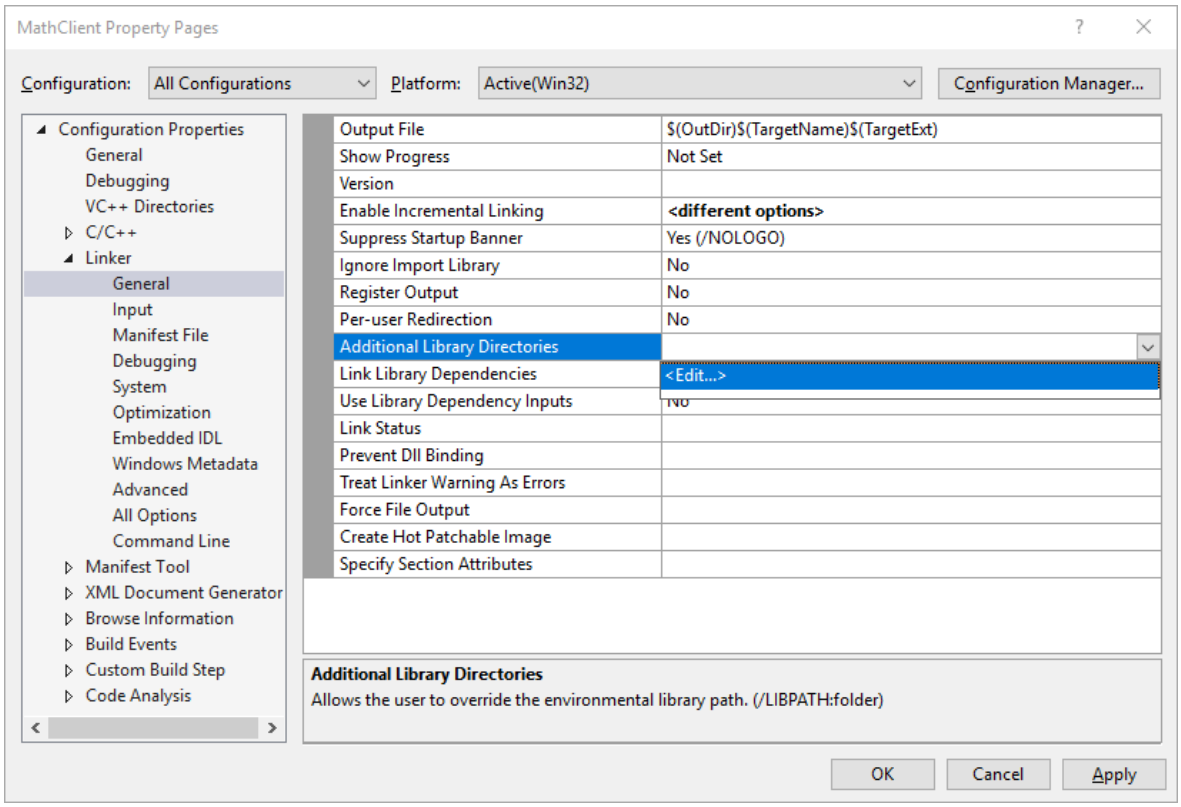


4. In the **Additional Dependencies** dialog, add *MathLibrary.lib* to the list in the top edit control.



5. Choose **OK** to go back to the **Property Pages** dialog box.

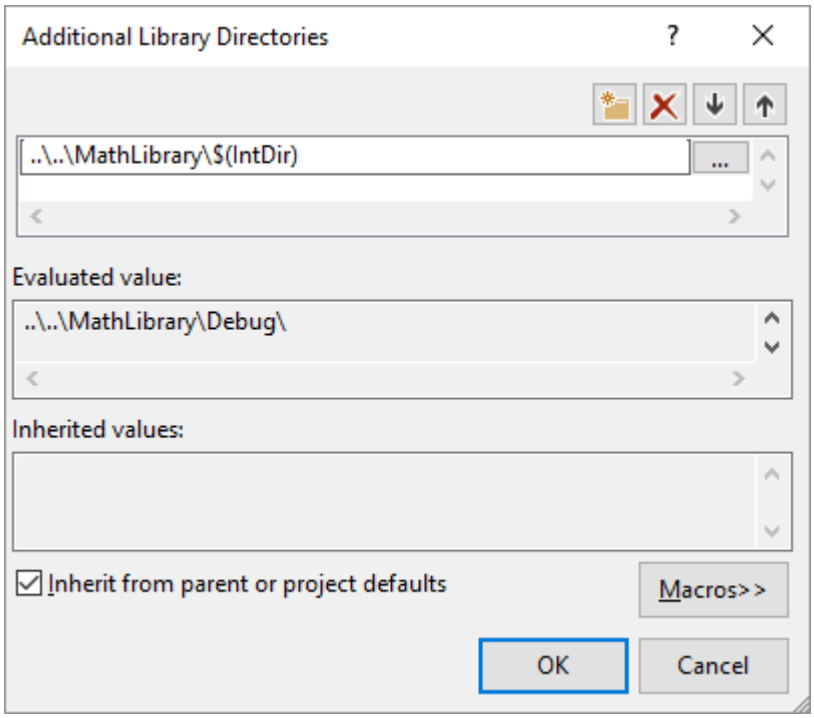
6. In the left pane, select **Configuration Properties** > **Linker** > **General**. In the property pane, select the drop-down control next to the **Additional Library Directories** edit box, and then choose **Edit**.



7. Double-click in the top pane of the **Additional Library Directories** dialog box to enable an edit control. In the edit control, specify the path to the location of the **MathLibrary.lib** file. By default, it's in a folder called *Debug* directly under the DLL solution folder. If you create a release build, the file is placed in a folder called *Release*. You can use the `$(IntDir)` macro so that the linker can find your DLL, no matter which kind of build you create. If you followed the directions to put your client project in a separate solution from the DLL project, the relative path should look like this:

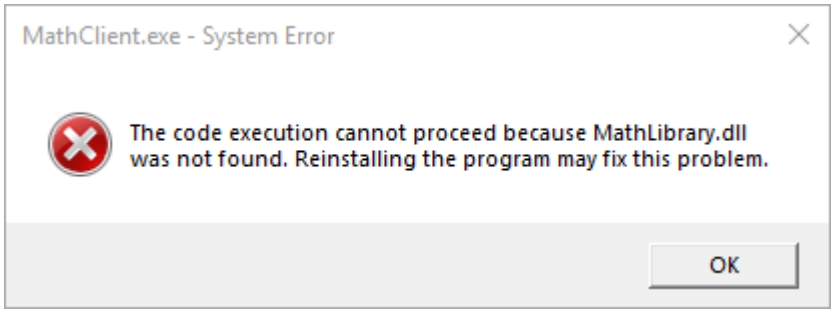
   ```
   ..\..\MathLibrary\$(IntDir)
   ```

 If your DLL and client projects are in other locations, adjust the relative path to match.



8. Once you've entered the path to the library file in the **Additional Library Directories** dialog box, choose the **OK** button to go back to the **Property Pages** dialog box. Choose **OK** to save the property changes.

Your client app can now compile and link successfully, but it still doesn't have everything it needs to run. When the operating system loads your app, it looks for the MathLibrary DLL. If it can't find the DLL in certain system directories, the environment path, or the local app directory, the load fails. Depending on the operating system, you'll see an error message like this:



One way to avoid this issue is to copy the DLL to the directory that contains your client executable as part of the build process. You can add a **Post-Build Event** to your project, to add a command that copies the DLL to your build output directory. The command specified here copies the DLL only if it's missing or has changed. It uses macros to copy to and from the Debug or Release locations, based on your build configuration.

To copy the DLL in a post-build event

1. Right-click on the **MathClient** node in **Solution Explorer** and choose **Properties** to open the **Property Pages** dialog.

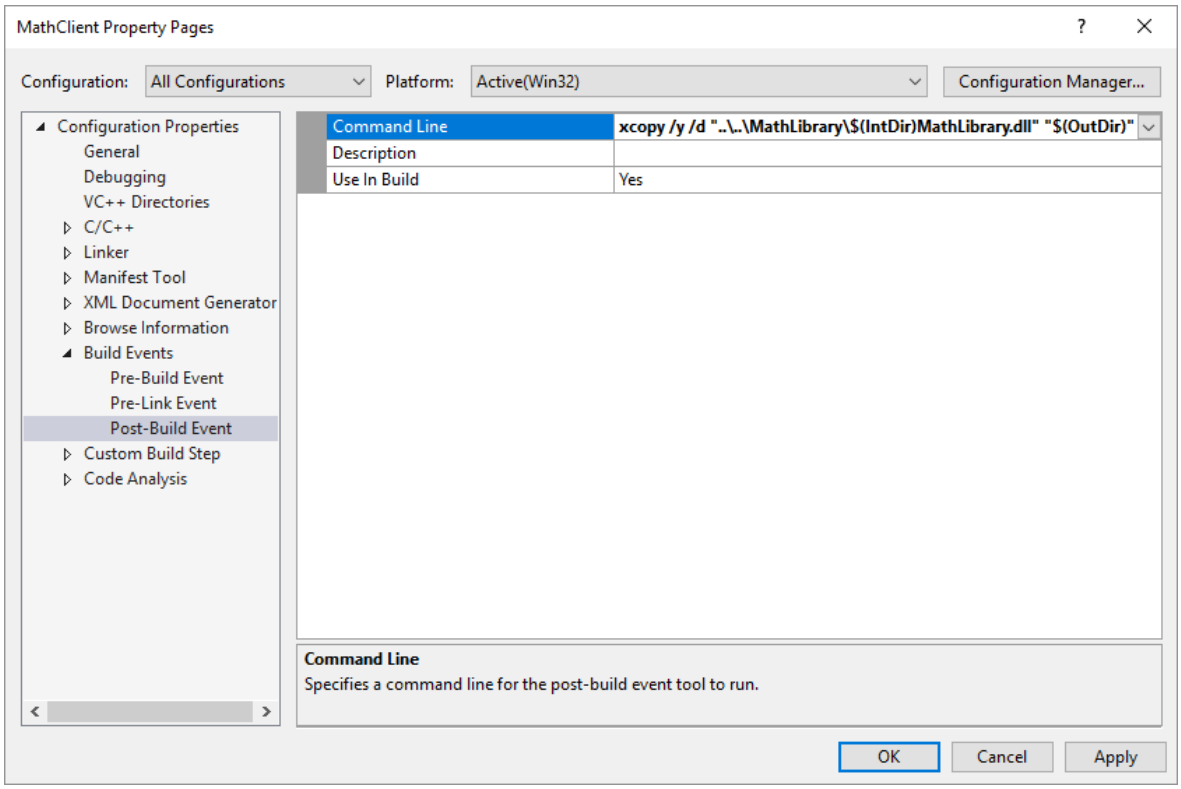
2. In the **Configuration** drop-down box, select **All Configurations** if it isn't already selected.

3. In the left pane, select **Configuration Properties** > **Build Events** > **Post-Build Event**.

4. In the property pane, select the edit control in the **Command Line** field. If you followed the directions to put your client project in a separate solution from the DLL project, then enter this command:

   ```
   xcopy /y /d "..\..\MathLibrary\$(IntDir)MathLibrary.dll" "$(OutDir)"
   ```

 If your DLL and client projects are in other directories, change the relative path to the DLL to match.



5. Choose the **OK** button to save your changes to the project properties.

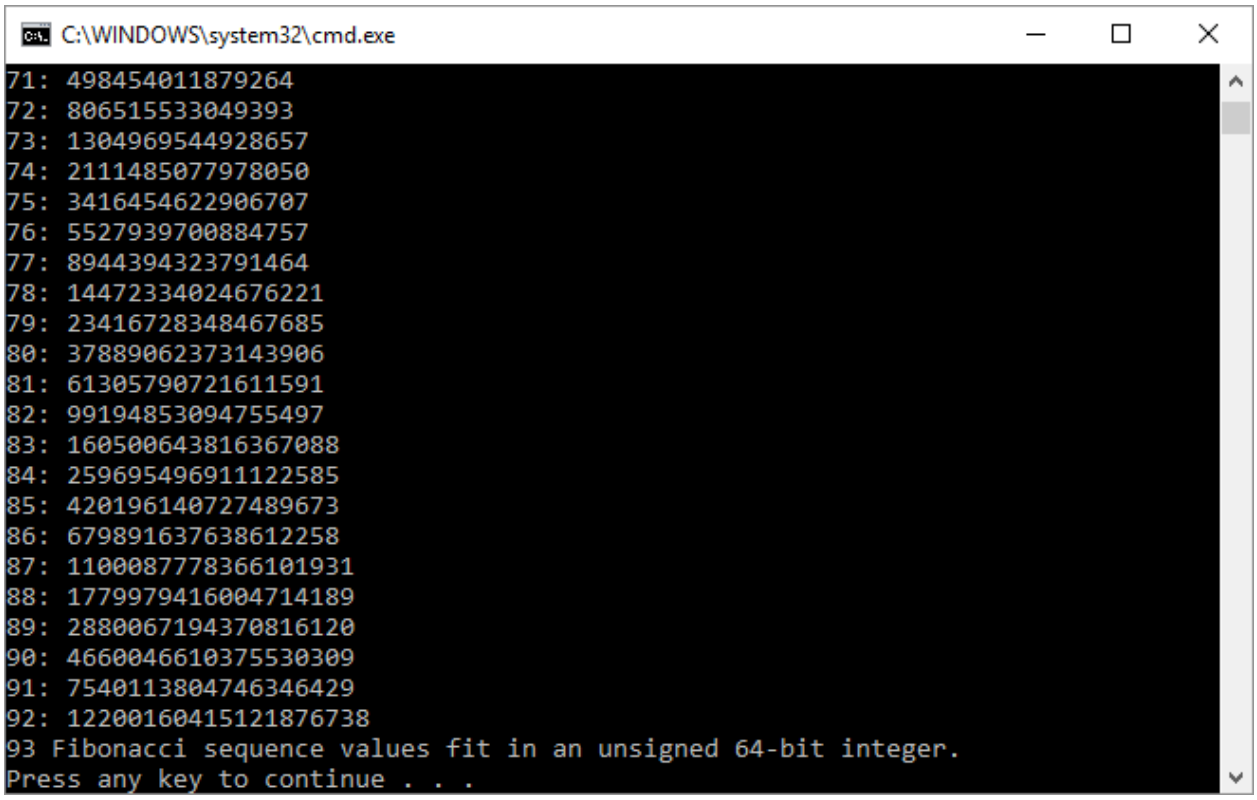
Now your client app has everything it needs to build and run. Build the application by choosing **Build** > **Build Solution** on the menu bar. The **Output** window in Visual Studio should have something like the following example depending on your version of Visual Studio:

Output

```
1>------ Build started: Project: MathClient, Configuration: Debug Win32 ----
--
1>MathClient.cpp
1>MathClient.vcxproj ->
```

```
C:\Users\username\Source\Repos\MathClient\Debug\MathClient.exe
1>1 File(s) copied
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Congratulations, you've created an application that calls functions in your DLL. Now run your application to see what it does. On the menu bar, choose **Debug** > **Start Without Debugging**. Visual Studio opens a command window for the program to run in. The last part of the output should look like:



Press any key to dismiss the command window.

Now that you've created a DLL and a client application, you can experiment. Try setting breakpoints in the code of the client app, and run the app in the debugger. See what happens when you step into a library call. Add other functions to the library, or write another client app that uses your DLL.

When you deploy your app, you must also deploy the DLLs it uses. The simplest way to make the DLLs that you build, or that you include from third parties, available is to put them in the same directory as your app. It's known as *app-local deployment*. For more information about deployment, see Deployment in Visual C++.

See also

Calling DLL Functions from Visual Basic Applications

Kinds of DLLs

Article • 08/03/2021

This topic provides information to help you determine the kind of DLL to build.

Different Kinds of DLLs Available

Using Visual Studio, you can build Win32 DLLs in C or C++ that do not use the Microsoft Foundation Class (MFC) library. You can create a non-MFC DLL project with the Win32 Application Wizard.

The MFC library itself is available, in either static link libraries or in a number of DLLs, with the MFC DLL Wizard. If your DLL is using MFC, Visual Studio supports three different DLL development scenarios:

- Building a regular MFC DLL that statically links MFC
- Building a regular MFC DLL that dynamically links MFC
- Building an MFC extension DLL, which always dynamically link MFC

What do you want to know more about?

- Non-MFC DLLs: Overview
- Regular MFC DLLs statically linked to MFC
- Regular MFC DLLs dynamically linked to MFC
- MFC extension DLLs: Overview
- Which kind of DLL to use

Deciding Which Kind of DLL to Use

If your DLL does not use MFC, use Visual Studio to build a non-MFC Win32 DLL. Linking your DLL to MFC (either statically or dynamically) takes up significant disk space and memory. You should not link to MFC unless your DLL actually uses MFC.

If your DLL will be using MFC, and will be used by either MFC or non-MFC applications, you must build either a regular MFC DLL that dynamically links to MFC or a regular MFC DLL that statically links to MFC. In most cases, you probably want to use a regular MFC

DLL that dynamically links to MFC because the file size of the DLL will be much smaller and the savings in memory from using the shared version of MFC can be significant. If you statically link to MFC, the file size of your DLL will be larger and potentially take up extra memory because it loads its own private copy of the MFC library code.

Building a DLL that dynamically links to MFC is faster than building a DLL that statically links to MFC because it is not necessary to link MFC itself. This is especially true in debug builds where the linker must compact the debug information. By linking with a DLL that already contains the debug information, there is less debug information to compact within your DLL.

One disadvantage to dynamically linking to MFC is that you must distribute the shared DLLs Mfcx0.dll and Msvcrxx.dll (or similar files) with your DLL. The MFC DLLs are freely redistributable, but you still must install the DLLs in your setup program. In addition, you must ship the Msvcrxx.dll, which contains the C run-time library that is used both by your program and the MFC DLLs themselves.

If your DLL will only be used by MFC executables, you have a choice between building a regular MFC DLL or an MFC extension DLL. If your DLL implements reusable classes derived from the existing MFC classes or you need to pass MFC-derived objects between the application and the DLL, you must build an MFC extension DLL.

If your DLL dynamically links to MFC, the MFC DLLs might be redistributed with your DLL. This architecture is particularly useful for sharing the class library between multiple executable files to save disk space and minimize memory usage.

What do you want to know more about?

- Non-MFC DLLs: Overview
- Regular MFC DLLs statically linked to MFC
- Regular MFC DLLs dynamically linked to MFC
- MFC extension DLLs: Overview

See also

Create C/C++ DLLs in Visual Studio

Non-MFC DLLs: Overview

Article • 08/03/2021

A non-MFC DLL is a DLL that does not use MFC internally, and the exported functions in the DLL can be called by either MFC or non-MFC executable files. Functions are usually exported from a non-MFC DLL using the standard C interface.

For more information about non-MFC DLLs, see Dynamic-Link Libraries in the Windows SDK.

What do you want to do?

- Walkthrough: Creating and Using a Dynamic Link Library
- Export from a DLL
- Link an executable to a DLL
- Initialize a DLL

What do you want to know more about?

- Regular MFC DLLs statically linked to MFC
- Regular MFC DLLs dynamically linked to MFC
- MFC extension DLLs: Overview

See also

Kinds of DLLs

Regular MFC DLLs Statically Linked to MFC

Article • 08/03/2021

A regular MFC DLL statically linked to MFC is a DLL that uses MFC internally, and the exported functions in the DLL can be called by either MFC or non-MFC executables. As the name describes, this kind of DLL is built using the static link library version of MFC. Functions are usually exported from a regular MFC DLL using the standard C interface. For an example of how to write, build, and use a regular MFC DLL, see the sample DLLScreenCap.

Note that the term USRDLL is no longer used in the Visual C++ documentation. A regular MFC DLL that is statically linked to MFC has the same characteristics as the former USRDLL.

A regular MFC DLL, statically linked to MFC, has the following features:

- The client executable can be written in any language that supports the use of DLLs (C, C++, Pascal, Visual Basic, and so on); it does not have to be an MFC application.
- The DLL can link to the same MFC static link libraries used by applications. There is no longer a separate version of the static link libraries for DLLs.
- Before version 4.0 of MFC, USRDLLs provided the same type of functionality as regular MFC DLLs statically linked to MFC. As of Visual C++ version 4.0, the term USRDLL is obsolete.

A regular MFC DLL, statically linked to MFC, has the following requirements:

- This type of DLL must instantiate a class derived from `CWinApp`.
- This type of DLL uses the `DllMain` provided by MFC. Place all DLL-specific initialization code in the `InitInstance` member function and termination code in `ExitInstance` as in a normal MFC application.
- Even though the term USRDLL is obsolete, you must still define "**_USRDLL**" on the compiler command line. This definition determines which declarations is pulled in from the MFC header files.

regular MFC DLLs must have a `CWinApp`-derived class and a single object of that application class, as does an MFC application. However, the `CWinApp` object of the DLL does not have a main message pump, as does the `CWinApp` object of an application.

Note that the `CWinApp::Run` mechanism does not apply to a DLL, because the application owns the main message pump. If the DLL opens modeless dialogs or has a main frame window of its own, the application's main message pump must call a routine exported by the DLL that in turn calls the `CWinApp::PreTranslateMessage` member function of the DLL's application object.

For an example of this function, see the DLLScreenCap sample.

Symbols are usually exported from a regular MFC DLL using the standard C interface. The declaration of a function exported from a regular MFC DLL would look something like this:

```
extern "C" __declspec(dllexport) MyExportedFunction( );
```

All memory allocations within a regular MFC DLL should stay within the DLL; the DLL should not pass to or receive from the calling executable any of the following:

- Pointers to MFC objects
- Pointers to memory allocated by MFC

If you need to do any of the above or need to pass MFC-derived objects between the calling executable and the DLL, you must build an MFC extension DLL.

It is safe to pass pointers to memory that were allocated by the C run-time libraries between an application and a DLL only if you make a copy of the data. You must not delete or resize these pointers or use them without making a copy of the memory.

A DLL that is statically linked to MFC cannot also dynamically link to the shared MFC DLLs. A DLL that is statically linked to MFC is dynamically bound to an application just like any other DLL; applications link to it just like any other DLL.

The standard MFC static link libraries are named according to the convention described in Naming Conventions for MFC DLLs. However, with MFC version 3.0 and later, it is no longer necessary to manually specify to the linker the version of the MFC library you want linked in. Instead, the MFC header files automatically determine the correct version of the MFC library to link in based on preprocessor defines, such as **_DEBUG** or **_UNICODE**. The MFC header files add /DEFAULTLIB directives instructing the linker to link in a specific version of the MFC library.

What do you want to do?

- Initialize regular MFC DLLs

What do you want to know more about?

- Using MFC as Part of a DLL
- Using Database, OLE, and Sockets MFC extension DLLs in regular MFC DLLs
- Creating an MFC DLL
- Regular MFC DLLs Dynamically Linked to MFC
- MFC extension DLLs

See also

Kinds of DLLs

Regular MFC DLLs Dynamically Linked to MFC

Article • 08/03/2021

A regular MFC DLL dynamically linked to MFC is a DLL that uses MFC internally, and the exported functions in the DLL can be called by either MFC or non-MFC executables. As the name describes, this kind of DLL is built using the dynamic-link library version of MFC (also known as the shared version of MFC). Functions are usually exported from a regular MFC DLL using the standard C interface.

You must add the `AFX_MANAGE_STATE` macro at the beginning of all the exported functions in regular MFC DLLs that dynamically link to MFC to set the current module state to the one for the DLL. This is done by adding the following line of code to the beginning of functions exported from the DLL:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

A regular MFC DLL, dynamically linked to MFC has the following features:

- This is a new type of DLL introduced by Visual C++ 4.0.
- The client executable can be written in any language that supports the use of DLLs (C, C++, Pascal, Visual Basic, and so on); it does not have to be an MFC application.
- Unlike the statically linked regular MFC DLL, this type of DLL is dynamically linked to the MFC DLL (also known as the shared MFC DLL).
- The MFC import library linked to this type of DLL is the same one used for MFC extension DLLs or applications using the MFC DLL: MFCxx(D).lib.

A regular MFC DLL, dynamically linked to MFC has the following requirements:

- These DLLs are compiled with **_AFXDLL** defined, just like an executable that is dynamically linked to the MFC DLL. But **_USRDLL** is also defined, just like a regular MFC DLL that is statically linked to MFC.
- This type of DLL must instantiate a `CWinApp`-derived class.
- This type of DLL uses the `DllMain` provided by MFC. Place all DLL-specific initialization code in the `InitInstance` member function and termination code in

`ExitInstance` as in a normal MFC application.

Because this kind of DLL uses the dynamic-link library version of MFC, you must explicitly set the current module state to the one for the DLL. To do this, use the AFX_MANAGE_STATE macro at the beginning of every function exported from the DLL.

regular MFC DLLs must have a `CWinApp`-derived class and a single object of that application class, as does an MFC application. However, the `CWinApp` object of the DLL does not have a main message pump, as does the `CWinApp` object of an application.

Note that the `CWinApp::Run` mechanism does not apply to a DLL, because the application owns the main message pump. If your DLL brings up modeless dialogs or has a main frame window of its own, your application's main message pump must call a DLL-exported routine that calls `CWinApp::PreTranslateMessage`.

Place all DLL-specific initialization in the `CWinApp::InitInstance` member function as in a normal MFC application. The `CWinApp::ExitInstance` member function of your `CWinApp` derived class is called from the MFC provided `DllMain` function before the DLL is unloaded.

You must distribute the shared DLLs MFCx0.dll and Msvcr*0.dll (or similar files) with your application.

A DLL that is dynamically linked to MFC cannot also statically link to MFC. Applications link to regular MFC DLLs dynamically linked to MFC it just like any other DLL.

Symbols are usually exported from a regular MFC DLL using the standard C interface. The declaration of a function exported from a regular MFC DLL looks something like this:

```
extern "C" __declspec(dllexport) MyExportedFunction( );
```

All memory allocations within a regular MFC DLL should stay within the DLL; the DLL should not pass to or receive from the calling executable any of the following:

- pointers to MFC objects
- pointers to memory allocated by MFC

If you need to do any of the above, or if you need to pass MFC-derived objects between the calling executable and the DLL, then you must build an MFC extension DLL.

It is safe to pass pointers to memory that were allocated by the C run-time libraries between an application and a DLL only if you make a copy of the data. You must not delete or resize these pointers or use them without making a copy of the memory.

When building a regular MFC DLL that dynamically links to MFC, you need to use the macro AFX_MANAGE_STATE to switch the MFC module state correctly. This is done by adding the following line of code to the beginning of functions exported from the DLL:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

The **AFX_MANAGE_STATE** macro should not be used in regular MFC DLLs that statically link to MFC or in MFC extension DLLs. For more information, see Managing the State Data of MFC Modules.

For an example of how to write, build, and use a regular MFC DLL, see the sample DLLScreenCap. For more information about regular MFC DLLs that dynamically link to MFC, see the section titled "Converting DLLScreenCap to Dynamically Link with the MFC DLL" in the abstract for the sample.

What do you want to do?

- Initialize regular MFC DLLs

What do you want to know more about?

- The module states of a regular MFC DLL dynamically linked to MFC
- Managing the state data of MFC modules
- Using Database, OLE, and Sockets MFC extension DLLs in regular MFC DLLs
- Using MFC as Part of a DLL

See also

Kinds of DLLs

MFC extension DLLs: Overview

Article • 08/03/2021

An MFC extension DLL is a DLL that typically implements reusable classes derived from existing Microsoft Foundation Class Library classes. MFC extension DLLs are built using the dynamic-link library version of MFC (also known as the shared version of MFC). Only MFC executables (either applications or regular MFC DLLs) that are built with the shared version of MFC can use an MFC extension DLL. With an MFC extension DLL, you can derive new custom classes from MFC and then offer this extended version of MFC to applications that call your DLL.

Extension DLLs can also be used for passing MFC-derived objects between the application and the DLL. The member functions associated with the passed object exist in the module where the object was created. Because these functions are properly exported when using the shared DLL version of MFC, you can freely pass MFC or MFC-derived object pointers between an application and the MFC extension DLLs it loads.

For an example of a DLL that fulfills the basic requirements of an MFC extension DLL, see the MFC sample DLLHUSK. In particular, look at the Testdll1.cpp and Testdll2.cpp files.

What do you want to do?

- Initialize an MFC extension DLL

What do you want to know more about?

- MFC extension DLLs
- Using Database, OLE, and Sockets MFC extension DLLs in regular MFC DLLs
- Non-MFC DLLs: Overview
- Regular MFC DLLs statically linked to MFC
- Regular MFC DLLs dynamically linked to MFC
- Creating an MFC DLL

See also

Kinds of DLLs

DLL frequently asked questions

FAQ

Can an MFC DLL create multiple threads?

Except during initialization, an MFC DLL can safely create multiple threads as long as it uses the Win32 thread local storage (TLS) functions such as **TlsAlloc** to allocate thread local storage. However, if an MFC DLL uses `__declspec(thread)` to allocate thread local storage, the client application must be implicitly linked to the DLL. If the client application explicitly links to the DLL, the call to **LoadLibrary** will not successfully load the DLL. For more information about thread-local variables in DLLs, see thread.

An MFC DLL that creates a new MFC thread during startup will stop responding when it is loaded by an application. This includes whenever a thread is created by calling `AfxBeginThread` or `CWinThread::CreateThread` inside:

- The `InitInstance` of a `CWinApp`-derived object in a regular MFC DLL.
- A supplied `DllMain` or **RawDllMain** function in a regular MFC DLL.
- A supplied `DllMain` or **RawDllMain** function in an MFC extension DLL.

Can a multithreaded application access an MFC DLL in different threads?

Multithreaded applications can access regular MFC DLLs that dynamically link to MFC and MFC extension DLLs from different threads. An application can access regular MFC DLLs that statically link to MFC from multiple threads created in the application.

Are there any MFC classes or functions that cannot be used in an MFC DLL?

Extension DLLs use the `CWinApp`-derived class of the client application. They must not have their own `CWinApp`-derived class.

Regular MFC DLLs must have a `CWinApp`-derived class and a single object of that application class, as does an MFC application. Unlike the `CWinApp` object of an application, the `CWinApp` object of the DLL does not have a main message pump.

Note that because the `CWinApp::Run` mechanism does not apply to a DLL, the application owns the main message pump. If the DLL opens modeless dialog boxes or has a main frame window of its own, the application's main message pump must call a routine exported by the DLL, which in turn calls the `CWinApp::PreTranslateMessage` member function of the DLL's application object.

What optimization techniques should I use to improve the client application's performance when loading?

If your DLL is a regular MFC DLL that is statically linked to MFC, changing it to a regular MFC DLL that is dynamically linked to MFC reduces the file size.

If the DLL has a large number of exported functions, use a .def file to export the functions (instead of using `__declspec(dllexport)`) and use the .def file NONAME attribute on each exported function. The NONAME attribute causes only the ordinal value and not the function name to be stored in the DLL's export table, which reduces the file size.

DLLs that are implicitly linked to an application are loaded when the application loads. To improve the performance when loading, try dividing the DLL into different DLLs. Put all the functions that the calling application needs immediately after loading into one DLL and have the calling application implicitly link to that DLL. Put the other functions that the calling application does not need right away into another DLL and have the application explicitly link to that DLL. For more information, see Link an executable to a DLL.

There's a memory leak in my regular MFC DLL, but my code looks fine. How can I find the memory leak?

One possible cause of the memory leak is that MFC creates temporary objects that are used inside message handler functions. In MFC applications, these temporary objects are automatically cleaned up in the `CWinApp::OnIdle()` function that is called in between

processing messages. However, in MFC dynamic-link libraries (DLLs), the `OnIdle()` function is not automatically called. As a result, temporary objects are not automatically cleaned up. To clean up temporary objects, the DLL must explicitly call `OnIdle(1)` periodically.

See also

Create C/C++ DLLs in Visual Studio

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Link an executable to a DLL

Article • 08/03/2021

An executable file links to (or loads) a DLL in one of two ways:

- *Implicit linking*, where the operating system loads the DLL at the same time as the executable that uses it. The client executable calls the exported functions of the DLL the same way as if the functions were statically linked and contained within the executable. Implicit linking is sometimes referred to as *static load* or *load-time dynamic linking*.
- *Explicit linking*, where the operating system loads the DLL on demand at runtime. An executable that uses a DLL by explicit linking must explicitly load and unload the DLL. It must also set up a function pointer to access each function it uses from the DLL. Unlike calls to functions in a statically linked library or an implicitly linked DLL, the client executable must call the exported functions in an explicitly linked DLL through function pointers. Explicit linking is sometimes referred to as *dynamic load* or *run-time dynamic linking*.

An executable can use either linking method to link to the same DLL. Furthermore, these methods aren't mutually exclusive; one executable may implicitly link to a DLL, and another might attach to it explicitly.

Determine which linking method to use

Whether to use implicit linking or explicit linking is an architectural decision you must make for your application. There are advantages and disadvantages to each method.

Implicit Linking

Implicit linking occurs when an application's code calls an exported DLL function. When the source code for the calling executable is compiled or assembled, the DLL function call generates an external function reference in the object code. To resolve this external reference, the application must link with the import library (.lib file) provided by the maker of the DLL.

The import library only contains code to load the DLL and to implement calls to functions in the DLL. Finding an external function in an import library informs the linker that the code for that function is in a DLL. To resolve external references to DLLs, the linker simply adds information to the executable file that tells the system where to find the DLL code when the process starts up.

When the system starts a program that contains dynamically linked references, it uses the information in the program's executable file to locate the required DLLs. If it can't locate the DLL, the system terminates the process, and displays a dialog box that reports the error. Otherwise, the system maps the DLL modules into the process address space.

If any of the DLLs has an entry-point function for initialization and termination code such as `DllMain`, the operating system calls the function. One of the parameters passed to the entry-point function specifies a code that indicates the DLL is attaching to the process. If the entry-point function doesn't return TRUE, the system terminates the process and reports the error.

Finally, the system modifies the executable code of the process to provide starting addresses for the DLL functions.

Like the rest of a program's code, the loader maps DLL code into the address space of the process when the process starts up. The operating system loads it into memory only when needed. As a result, the `PRELOAD` and `LOADONCALL` code attributes used by .def files to control loading in previous versions of Windows no longer have meaning.

Explicit Linking

Most applications use implicit linking because it's the easiest linking method to use. However, there are times when explicit linking is necessary. Here are some common reasons to use explicit linking:

- The application doesn't know the name of a DLL that it loads until run time. For example, the application might obtain the name of the DLL and the exported functions from a configuration file at startup.
- A process that uses implicit linking is terminated by the operating system if the DLL isn't found at process startup. A process that uses explicit linking isn't terminated in this situation, and can attempt to recover from the error. For example, the process could notify the user of the error and have the user specify another path to the DLL.
- A process that uses implicit linking is also terminated if any of the DLLs it's linked to have a `DllMain` function that fails. A process that uses explicit linking isn't terminated in this situation.
- An application that implicitly links to many DLLs can be slow to start because Windows loads all the DLLs when the application loads. To improve startup performance, an application might only use implicit linking for DLLs required

immediately after loading. It might use explicit linking to load other DLLs only when they're needed.

- Explicit linking eliminates the need to link the application by using an import library. If changes in the DLL cause the export ordinals to change, applications don't have to relink if they call `GetProcAddress` using the name of a function and not an ordinal value. Applications that use implicit linking must still relink to the changed import library.

Here are two hazards of explicit linking to be aware of:

- If the DLL has a `DllMain` entry point function, the operating system calls the function in the context of the thread that called `LoadLibrary`. The entry-point function isn't called if the DLL is already attached to the process because of a previous call to `LoadLibrary` that has had no corresponding call to the `FreeLibrary` function. Explicit linking can cause problems if the DLL uses a `DllMain` function to initialize each thread of a process, because any threads that already exist when `LoadLibrary` (or `AfxLoadLibrary`) is called aren't initialized.

- If a DLL declares static-extent data as `__declspec(thread)`, it can cause a protection fault if explicitly linked. After the DLL is loaded by a call to `LoadLibrary`, it causes a protection fault whenever the code references this data. (Static-extent data includes both global and local static items.) That's why, when you create a DLL, you should avoid using thread-local storage. If you can't, then inform your DLL users about the potential pitfalls of dynamically loading your DLL. For more information, see Using thread local storage in a dynamic-link library (Windows SDK).

How to use implicit linking

To use a DLL by implicit linking, client executables must obtain these files from the provider of the DLL:

- One or more header files (.h files) that contain the declarations of the exported data, functions, and C++ classes in the DLL. The classes, functions, and data exported by the DLL must all be marked `__declspec(dllimport)` in the header file. For more information, see dllexport, dllimport.

- An import library to link into your executable. The linker creates the import library when the DLL is built. For more information, see LIB files as linker input.

- The actual DLL file.

To use the data, functions, and classes in a DLL by implicit linking, any client source file must include the header files that declare them. From a coding perspective, calls to the exported functions are just like any other function call.

To build the client executable file, you must link with the DLL's import library. If you use an external makefile or build system, specify the import library together with the other object files or libraries that you link.

The operating system must be able to locate the DLL file when it loads the calling executable. That means you must either deploy or verify the existence of the DLL when you install your application.

How to link explicitly to a DLL

To use a DLL by explicit linking, applications must make a function call to explicitly load the DLL at run time. To explicitly link to a DLL, an application must:

- Call LoadLibraryEx or a similar function to load the DLL and obtain a module handle.
- Call GetProcAddress to obtain a function pointer to each exported function that the application calls. Because applications call the DLL functions through a pointer, the compiler doesn't generate external references, so there's no need to link with an import library. However, you must have a `typedef` or `using` statement that defines the call signature of the exported functions that you call.
- Call FreeLibrary when done with the DLL.

For example, this sample function calls `LoadLibrary` to load a DLL named "MyDLL", calls `GetProcAddress` to obtain a pointer to a function named "DLLFunc1", calls the function and saves the result, and then calls `FreeLibrary` to unload the DLL.

C

```
#include "windows.h"

typedef HRESULT (CALLBACK* LPFNDLLFUNC1)(DWORD,UINT*);

HRESULT LoadAndCallSomeFunction(DWORD dwParam1, UINT * puParam2)
{
    HINSTANCE hDLL;               // Handle to DLL
    LPFNDLLFUNC1 lpfnDllFunc1;    // Function pointer
    HRESULT hrReturnVal;

    hDLL = LoadLibrary("MyDLL");
    if (NULL != hDLL)
```

```
    {
        lpfnDllFunc1 = (LPFNDLLFUNC1)GetProcAddress(hDLL, "DLLFunc1");
        if (NULL != lpfnDllFunc1)
        {
            // call the function
            hrReturnVal = lpfnDllFunc1(dwParam1, puParam2);
        }
        else
        {
            // report the error
            hrReturnVal = ERROR_DELAY_LOAD_FAILED;
        }
        FreeLibrary(hDLL);
    }
    else
    {
        hrReturnVal = ERROR_DELAY_LOAD_FAILED;
    }
    return hrReturnVal;
}
```

Unlike this example, in most cases you should call `LoadLibrary` and `FreeLibrary` only once in your application for a given DLL. It's especially true if you're going to call multiple functions in the DLL, or call DLL functions repeatedly.

What do you want to know more about?

- Working with Import Libraries and Export Files
- Dynamic-Link Library Search Order

See also

Create C/C++ DLLs in Visual Studio

DLLs and Visual C++ run-time library behavior

Article • 03/02/2022

When you build a Dynamic-link Library (DLL) by using Visual Studio, by default, the linker includes the Visual C++ run-time library (VCRuntime). The VCRuntime contains code required to initialize and terminate a C/C++ executable. When linked into a DLL, the VCRuntime code provides an internal DLL entry-point function called `_DllMainCRTStartup` that handles Windows OS messages to the DLL to attach to or detach from a process or thread. The `_DllMainCRTStartup` function performs essential tasks such as stack buffer security set up, C run-time library (CRT) initialization and termination, and calls to constructors and destructors for static and global objects. `_DllMainCRTStartup` also calls hook functions for other libraries such as WinRT, MFC, and ATL to perform their own initialization and termination. Without this initialization, the CRT and other libraries, as well as your static variables, would be left in an uninitialized state. The same VCRuntime internal initialization and termination routines are called whether your DLL uses a statically linked CRT or a dynamically linked CRT DLL.

Default DLL entry point _DllMainCRTStartup

In Windows, all DLLs can contain an optional entry-point function, usually called `DllMain`, that is called for both initialization and termination. This gives you an opportunity to allocate or release additional resources as needed. Windows calls the entry-point function in four situations: process attach, process detach, thread attach, and thread detach. When a DLL is loaded into a process address space, either when an application that uses it is loaded, or when the application requests the DLL at runtime, the operating system creates a separate copy of the DLL data. This is called *process attach*. *Thread attach* occurs when the process the DLL is loaded in creates a new thread. *Thread detach* occurs when the thread terminates, and *process detach* is when the DLL is no longer required and is released by an application. The operating system makes a separate call to the DLL entry point for each of these events, passing a *reason* argument for each event type. For example, the OS sends `DLL_PROCESS_ATTACH` as the *reason* argument to signal process attach.

The VCRuntime library provides an entry-point function called `_DllMainCRTStartup` to handle default initialization and termination operations. On process attach, the `_DllMainCRTStartup` function sets up buffer security checks, initializes the CRT and other libraries, initializes run-time type information, initializes and calls constructors for static

and non-local data, initializes thread-local storage, increments an internal static counter for each attach, and then calls a user- or library-supplied `DllMain`. On process detach, the function goes through these steps in reverse. It calls `DllMain`, decrements the internal counter, calls destructors, calls CRT termination functions and registered `atexit` functions, and notifies any other libraries of termination. When the attachment counter goes to zero, the function returns `FALSE` to indicate to Windows that the DLL can be unloaded. The `_DllMainCRTStartup` function is also called during thread attach and thread detach. In these cases, the VCRuntime code does no additional initialization or termination on its own, and just calls `DllMain` to pass the message along. If `DllMain` returns `FALSE` from process attach, signaling failure, `_DllMainCRTStartup` calls `DllMain` again and passes `DLL_PROCESS_DETACH` as the *reason* argument, then goes through the rest of the termination process.

When building DLLs in Visual Studio, the default entry point `_DllMainCRTStartup` supplied by VCRuntime is linked in automatically. You do not need to specify an entry-point function for your DLL by using the /ENTRY (Entry point symbol) linker option.

> **Note**
>
> While it is possible to specify another entry-point function for a DLL by using the /ENTRY: linker option, we do not recommend it, because your entry-point function would have to duplicate everything that `_DllMainCRTStartup` does, in the same order. The VCRuntime provides functions that allow you to duplicate its behavior. For example, you can call **__security_init_cookie** immediately on process attach to support the **/GS (Buffer security check)** buffer checking option. You can call the `_CRT_INIT` function, passing the same parameters as the entry point function, to perform the rest of the DLL initialization or termination functions.

Initialize a DLL

Your DLL may have initialization code that must execute when your DLL loads. In order for you to perform your own DLL initialization and termination functions, `_DllMainCRTStartup` calls a function called `DllMain` that you can provide. Your `DllMain` must have the signature required for a DLL entry point. The default entry point function `_DllMainCRTStartup` calls `DllMain` using the same parameters passed by Windows. By default, if you do not provide a `DllMain` function, Visual Studio provides one for you and links it in so that `_DllMainCRTStartup` always has something to call. This means that if you do not need to initialize your DLL, there is nothing special you have to do when building your DLL.

This is the signature used for `DllMain`:

```
C++

#include <windows.h>

extern "C" BOOL WINAPI DllMain (
    HINSTANCE const instance,  // handle to DLL module
    DWORD     const reason,    // reason for calling function
    LPVOID    const reserved); // reserved
```

Some libraries wrap the `DllMain` function for you. For example, in a regular MFC DLL, implement the `CWinApp` object's `InitInstance` and `ExitInstance` member functions to perform initialization and termination required by your DLL. For more details, see the Initialize regular MFC DLLs section.

> ⚠ **Warning**
>
> There are significant limits on what you can safely do in a DLL entry point. For more information about specific Windows APIs that are unsafe to call in `DllMain`, see **General Best Practices**. If you need anything but the simplest initialization then do that in an initialization function for the DLL. You can require applications to call the initialization function after `DllMain` has run and before they call any other functions in the DLL.

Initialize ordinary (non-MFC) DLLs

To perform your own initialization in ordinary (non-MFC) DLLs that use the VCRuntime-supplied `_DllMainCRTStartup` entry point, your DLL source code must contain a function called `DllMain`. The following code presents a basic skeleton showing what the definition of `DllMain` might look like:

```
C++

#include <windows.h>

extern "C" BOOL WINAPI DllMain (
    HINSTANCE const instance,  // handle to DLL module
    DWORD     const reason,    // reason for calling function
    LPVOID    const reserved)  // reserved
{
    // Perform actions based on the reason for calling.
    switch (reason)
    {
```

```
    case DLL_PROCESS_ATTACH:
        // Initialize once for each new process.
        // Return FALSE to fail DLL load.
        break;

    case DLL_THREAD_ATTACH:
        // Do thread-specific initialization.
        break;

    case DLL_THREAD_DETACH:
        // Do thread-specific cleanup.
        break;

    case DLL_PROCESS_DETACH:
        // Perform any necessary cleanup.
        break;
    }
    return TRUE;  // Successful DLL_PROCESS_ATTACH.
}
```

> ⓘ Note
>
> Older Windows SDK documentation says that the actual name of the DLL entry-point function must be specified on the linker command-line with the /ENTRY option. With Visual Studio, you do not need to use the /ENTRY option if the name of your entry-point function is `DllMain`. In fact, if you use the /ENTRY option and name your entry-point function something other than `DllMain`, the CRT does not get initialized properly unless your entry-point function makes the same initialization calls that `_DllMainCRTStartup` makes.

Initialize regular MFC DLLs

Because regular MFC DLLs have a `CWinApp` object, they should perform their initialization and termination tasks in the same location as an MFC application: in the `InitInstance` and `ExitInstance` member functions of the DLL's `CWinApp`-derived class. Because MFC provides a `DllMain` function that is called by `_DllMainCRTStartup` for `DLL_PROCESS_ATTACH` and `DLL_PROCESS_DETACH`, you should not write your own `DllMain` function. The MFC-provided `DllMain` function calls `InitInstance` when your DLL is loaded and it calls `ExitInstance` before the DLL is unloaded.

A regular MFC DLL can keep track of multiple threads by calling TlsAlloc and TlsGetValue in its `InitInstance` function. These functions allow the DLL to track thread-specific data.

In your regular MFC DLL that dynamically links to MFC, if you are using any MFC OLE, MFC Database (or DAO), or MFC Sockets support, respectively, the debug MFC extension DLLs MFCO*version*D.dll, MFCD*version*D.dll, and MFCN*version*D.dll (where *version* is the version number) are linked in automatically. You must call one of the following predefined initialization functions for each of these DLLs that you are using in your regular MFC DLL's `CWinApp::InitInstance`.

Type of MFC support	Initialization function to call
MFC OLE (MFCO*version*D.dll)	`AfxOleInitModule`
MFC Database (MFCD*version*D.dll)	`AfxDbInitModule`
MFC Sockets (MFCN*version*D.dll)	`AfxNetInitModule`

Initialize MFC extension DLLs

Because MFC extension DLLs do not have a `CWinApp`-derived object (as do regular MFC DLLs), you should add your initialization and termination code to the `DllMain` function that the MFC DLL Wizard generates.

The wizard provides the following code for MFC extension DLLs. In the code, `PROJNAME` is a placeholder for the name of your project.

C++

```
#include "pch.h" // For Visual Studio 2017 and earlier, use "stdafx.h"
#include <afxdllx.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
static AFX_EXTENSION_MODULE PROJNAMEDLL;

extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      TRACE0("PROJNAME.DLL Initializing!\n");

      // MFC extension DLL one-time initialization
      AfxInitExtensionModule(PROJNAMEDLL,
                             hInstance);

      // Insert this DLL into the resource chain
```

```
        new CDynLinkLibrary(Dll3DLL);
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        TRACE0("PROJNAME.DLL Terminating!\n");
    }
    return 1;   // ok
}
```

Creating a new `CDynLinkLibrary` object during initialization allows the MFC extension DLL to export `CRuntimeClass` objects or resources to the client application.

If you are going to use your MFC extension DLL from one or more regular MFC DLLs, you must export an initialization function that creates a `CDynLinkLibrary` object. That function must be called from each of the regular MFC DLLs that use the MFC extension DLL. An appropriate place to call this initialization function is in the `InitInstance` member function of the regular MFC DLL's `CWinApp`-derived object before using any of the MFC extension DLL's exported classes or functions.

In the `DllMain` that the MFC DLL Wizard generates, the call to `AfxInitExtensionModule` captures the module's run-time classes (`CRuntimeClass` structures) as well as its object factories (`COleObjectFactory` objects) for use when the `CDynLinkLibrary` object is created. You should check the return value of `AfxInitExtensionModule`; if a zero value is returned from `AfxInitExtensionModule`, return zero from your `DllMain` function.

If your MFC extension DLL will be explicitly linked to an executable (meaning the executable calls `AfxLoadLibrary` to link to the DLL), you should add a call to `AfxTermExtensionModule` on `DLL_PROCESS_DETACH`. This function allows MFC to clean up the MFC extension DLL when each process detaches from the MFC extension DLL (which happens when the process exits or when the DLL is unloaded as a result of a `AfxFreeLibrary` call). If your MFC extension DLL will be linked implicitly to the application, the call to `AfxTermExtensionModule` is not necessary.

Applications that explicitly link to MFC extension DLLs must call `AfxTermExtensionModule` when freeing the DLL. They should also use `AfxLoadLibrary` and `AfxFreeLibrary` (instead of the Win32 functions `LoadLibrary` and `FreeLibrary`) if the application uses multiple threads. Using `AfxLoadLibrary` and `AfxFreeLibrary` ensures that the startup and shutdown code that executes when the MFC extension DLL is loaded and unloaded does not corrupt the global MFC state.

Because the MFCx0.dll is fully initialized by the time `DllMain` is called, you can allocate memory and call MFC functions within `DllMain` (unlike the 16-bit version of MFC).

Extension DLLs can take care of multithreading by handling the `DLL_THREAD_ATTACH` and `DLL_THREAD_DETACH` cases in the `DllMain` function. These cases are passed to `DllMain` when threads attach and detach from the DLL. Calling TlsAlloc when a DLL is attaching allows the DLL to maintain thread local storage (TLS) indexes for every thread attached to the DLL.

Note that the header file Afxdllx.h contains special definitions for structures used in MFC extension DLLs, such as the definition for `AFX_EXTENSION_MODULE` and `CDynLinkLibrary`. You should include this header file in your MFC extension DLL.

> **Note**
>
> It is important that you neither define nor undefine any of the `_AFX_NO_XXX` macros in *pch.h* (*stdafx.h* in Visual Studio 2017 and earlier). These macros exist only for the purpose of checking whether a particular target platform supports that feature or not. You can write your program to check these macros (for example, `#ifndef _AFX_NO_OLE_SUPPORT`), but your program should never define or undefine these macros.

A sample initialization function that handles multithreading is included in Using Thread Local Storage in a Dynamic-Link Library in the Windows SDK. Note that the sample contains an entry-point function called `LibMain`, but you should name this function `DllMain` so that it works with the MFC and C run-time libraries.

See also

Create C/C++ DLLs in Visual Studio
DllMain entry point
Dynamic-link Library Best Practices

LoadLibrary and AfxLoadLibrary

Article • 08/03/2021

Processes call LoadLibrary or LoadLibraryEx to explicitly link to a DLL. (MFC apps use AfxLoadLibrary or AfxLoadLibraryEx.) If the function succeeds, it maps the specified DLL into the address space of the calling process, and returns a handle to the DLL. The handle is required in other functions used for explicit linking—for example, `GetProcAddress` and `FreeLibrary`. For more information, see Explicit linking.

`LoadLibrary` attempts to locate the DLL by using the same search sequence that is used for implicit linking. `LoadLibraryEx` gives you more control over the search path order. For more information, see Dynamic Link Library Search Order. If the system can't find the DLL or if the entry-point function returns FALSE, `LoadLibrary` returns NULL. If the call to `LoadLibrary` specifies a DLL module that is already mapped into the address space of the calling process, the function returns a handle of the DLL and increments the reference count of the module.

If the DLL has an entry-point function, the operating system calls the function in the context of the thread that called `LoadLibrary` or `LoadLibraryEx`. The entry-point function isn't called if the DLL is already attached to the process. That happens when a previous call to `LoadLibrary` or `LoadLibraryEx` for the DLL hasn't had a corresponding call to the `FreeLibrary` function.

For MFC applications that load MFC extension DLLs, we recommend that you use `AfxLoadLibrary` or `AfxLoadLibraryEx` instead of `LoadLibrary` or `LoadLibraryEx`. The MFC functions handle thread synchronization before loading the DLL explicitly. The interfaces (function prototypes) to `AfxLoadLibrary` and `AfxLoadLibraryEx` are the same as `LoadLibrary` and `LoadLibraryEx`.

If Windows can't load the DLL, your process can attempt to recover from the error. For example, it could notify the user of the error, then ask for another path to the DLL.

> **ⓘ Important**
>
> Make sure to specify the full path of any DLLs. The current directory may be searched first when files are loaded by `LoadLibrary`. If you don't fully qualify the path of the file, a file other than the intended one might be loaded. When you create a DLL, use the **/DEPENDENTLOADFLAG** linker option to specify a search order for statically linked DLL dependencies. Within your DLLs, use both complete paths to explicitly loaded dependencies, and `LoadLibraryEx` or `AfxLoadLibraryEx`

call parameters to specify module search order. For more information, see **Dynamic-Link Library Security** and **Dynamic Link Library Search Order**.

What do you want to do?

- Link an executable to a DLL
- Link an executable to a DLL

What do you want to know more about?

- Dynamic-Link Library Search Order
- FreeLibrary and AfxFreeLibrary
- GetProcAddress

See also

- Create C/C++ DLLs in Visual Studio

GetProcAddress

Article • 08/03/2021

Processes explicitly linking to a DLL call GetProcAddress to obtain the address of an exported function in the DLL. You use the returned function pointer to call the DLL function. **GetProcAddress** takes as parameters the DLL module handle (returned by either **LoadLibrary**, `AfxLoadLibrary`, or **GetModuleHandle**) and takes either the name of the function you want to call or the function's export ordinal.

Because you are calling the DLL function through a pointer and there is no compile-time type checking, make sure that the parameters to the function are correct so that you do not overstep the memory allocated on the stack and cause an access violation. One way to help provide type-safety is to look at the function prototypes of the exported functions and create matching typedefs for the function pointers. For example:

```
typedef UINT (CALLBACK* LPFNDLLFUNC1)(DWORD,UINT);
...

HINSTANCE hDLL;               // Handle to DLL
LPFNDLLFUNC1 lpfnDllFunc1;    // Function pointer
DWORD dwParam1;
UINT  uParam2, uReturnVal;

hDLL = LoadLibrary("MyDLL");
if (hDLL != NULL)
{
   lpfnDllFunc1 = (LPFNDLLFUNC1)GetProcAddress(hDLL,
                                               "DLLFunc1");
   if (!lpfnDllFunc1)
   {
      // handle the error
      FreeLibrary(hDLL);
      return SOME_ERROR_CODE;
   }
   else
   {
      // call the function
      uReturnVal = lpfnDllFunc1(dwParam1, uParam2);
   }
}
```

How you specify the function you want when calling **GetProcAddress** depends on how the DLL was built.

You can only obtain the export ordinal if the DLL you are linking to is built with a module definition (.def) file and if the ordinals are listed with the functions in the **EXPORTS** section of the DLL's .def file. Calling **GetProcAddress** with an export ordinal, as opposed to the function name, is slightly faster if the DLL has many exported functions because the export ordinals serve as indexes into the DLL's export table. With an export ordinal, **GetProcAddress** can locate the function directly as opposed to comparing the specified name to the function names in the DLL's export table. However, you should call **GetProcAddress** with an export ordinal only if you have control over assigning the ordinals to the exported functions in the .def file.

What do you want to do?

- Link an executable to a DLL
- Link an executable to a DLL

What do you want to know more about?

- LoadLibrary and AfxLoadLibrary
- FreeLibrary
- Exporting from a DLL Using DEF Files

See also

Create C/C++ DLLs in Visual Studio

FreeLibrary and AfxFreeLibrary

Article • 08/03/2021

Processes that explicitly link to a DLL call the FreeLibrary function when the DLL module is no longer needed. This function decrements the module's reference count. And, if the reference count is zero, it's unmapped from the address space of the process.

In an MFC application, use AfxFreeLibrary instead of `FreeLibrary` to unload an MFC extension DLL. The interface (function prototype) for `AfxFreeLibrary` is the same as `FreeLibrary`.

What do you want to do?

- Link an executable to a DLL
- Link an executable to a DLL

What do you want to know more about?

- LoadLibrary and AfxLoadLibrary
- GetProcAddress

See also

Create C/C++ DLLs in Visual Studio
FreeLibrary
AfxFreeLibrary

Module States of a Regular MFC DLL Dynamically Linked to MFC

Article • 08/03/2021

The ability to dynamically link a regular MFC DLL to the MFC DLL allows some configurations that are very complicated. For example, a regular MFC DLL and the executable that uses it can both dynamically link to the MFC DLL and to any MFC extension DLLs.

This configuration poses a problem with regard to the MFC global data, such as the pointer to the current `CWinApp` object and handle maps.

Before MFC version 4.0, this global data resided in the MFC DLL itself and was shared by all the modules in the process. Because each process using a Win32 DLL gets its own copy of the DLL's data, this scheme provided an easy way to track per-process data. Also, because the AFXDLL model presumed that there would be only one `CWinApp` object and only one set of handle maps in the process, these items could be tracked in the MFC DLL itself.

But with the ability to dynamically link a regular MFC DLL to the MFC DLL, it is now possible to have two or more `CWinApp` objects in a process — and also two or more sets of handle maps. How does MFC keep track of which ones it should be using?

The solution is to give each module (application or regular MFC DLL) its own copy of this global state information. Thus, a call to **AfxGetApp** in the regular MFC DLL returns a pointer to the `CWinApp` object in the DLL, not the one in the executable. This per-module copy of the MFC global data is known as a module state and is described in MFC Tech Note 58.

The MFC common window procedure automatically switches to the correct module state, so you do not need to worry about it in any message handlers implemented in your regular MFC DLL. But when your executable calls into the regular MFC DLL, you do need to explicitly set the current module state to the one for the DLL. To do this, use the **AFX_MANAGE_STATE** macro in every function exported from the DLL. This is done by adding the following line of code to the beginning of functions exported from the DLL:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState( ))
```

What do you want to know more about?

- Managing the state data of MFC modules
- Regular MFC DLLs dynamically linked to MFC
- MFC extension DLLs

See also

Create C/C++ DLLs in Visual Studio

MFC extension DLLs

Article • 08/03/2021

An MFC extension DLL is a DLL that typically implements reusable classes derived from the existing Microsoft Foundation Class Library classes.

An MFC extension DLL has the following features and requirements:

- The client executable must be an MFC application compiled with `_AFXDLL` defined.
- An MFC extension DLL can also be used by a regular MFC DLL that is dynamically linked to MFC.
- MFC extension DLLs should be compiled with `_AFXEXT` defined. This forces `_AFXDLL` to be also defined and ensures that the proper declarations is pulled in from the MFC header files. It also ensures that `AFX_EXT_CLASS` is defined as `__declspec(dllexport)` while building the DLL, which is necessary if you are using this macro to declare the classes in your MFC extension DLL.
- MFC extension DLLs should not instantiate a class derived from `CWinApp`, but should rely on the client application (or DLL) to provide this object.
- MFC extension DLLs should, however, provide a `DllMain` function and do any necessary initialization there.

Extension DLLs are built using the dynamic-link library version of MFC (also known as the shared version of MFC). Only MFC executables (either applications or regular MFC DLLs) that are built with the shared version of MFC can use an MFC extension DLL. Both the client application and the MFC extension DLL must use the same version of MFCx0.dll. With an MFC extension DLL, you can derive new custom classes from MFC and then offer this extended version of MFC to applications that call your DLL.

Extension DLLs can also be used for passing MFC-derived objects between the application and the DLL. The member functions associated with the passed object exist in the module where the object was created. Because these functions are properly exported when using the shared DLL version of MFC, you can freely pass MFC or MFC-derived object pointers between an application and the MFC extension DLLs it loads.

An MFC extension DLL uses a shared version of MFC in the same way an application uses the shared DLL version of MFC, with a few additional considerations:

- It does not have a `CWinApp`-derived object. It must work with the `CWinApp`-derived object of the client application. This means that the client application owns the

main message pump, the idle loop, and so on.

- It calls `AfxInitExtensionModule` in its `DllMain` function. The return value of this function should be checked. If a zero value is returned from `AfxInitExtensionModule`, return 0 from your `DllMain` function.

- It creates a **CDynLinkLibrary** object during initialization if the MFC extension DLL wants to export `CRuntimeClass` objects or resources to the application.

Before version 4.0 of MFC, this type of DLL was called an AFXDLL. AFXDLL refers to the `_AFXDLL` preprocessor symbol that is defined when building the DLL.

The import libraries for the shared version of MFC are named according to the convention described in Naming conventions for MFC DLLs. Visual Studio supplies prebuilt versions of the MFC DLLs, plus a number of non-MFC DLLs that you can use and distribute with your applications. These are documented in Redist.txt, which is installed to the Program Files\Microsoft Visual Studio folder.

If you are exporting using a .def file, place the following code at the beginning and end of your header file:

C++

```
#undef AFX_DATA
#define AFX_DATA AFX_EXT_DATA
// <body of your header file>
#undef AFX_DATA
#define AFX_DATA
```

These four lines ensure that your code is compiled correctly for an MFC extension DLL. Leaving out these four lines might cause your DLL to either compile or link incorrectly.

If you need to pass an MFC or MFC-derived object pointer to or from an MFC DLL, the DLL should be an MFC extension DLL. The member functions associated with the passed object exist in the module where the object was created. Because these functions are properly exported when using the shared DLL version of MFC, you can freely pass MFC or MFC-derived object pointers between an application and the MFC extension DLLs it loads.

Due to C++ name mangling and export issues, the export list from an MFC extension DLL might be different between the debug and retail versions of the same DLL and DLLs for different platforms. The retail MFCx0.dll has about 2,000 exported entry points; the debug MFCx0D.dll has about 3,000 exported entry points.

Memory Management

MFCx0.dll and all MFC extension DLLs loaded into a client application's address space use the same memory allocator, resource loading, and other MFC global states as if they were in the same application. This is significant because the non-MFC DLL libraries and the regular MFC DLLs do the exact opposite and have each DLL allocating out of its own memory pool.

If an MFC extension DLL allocates memory, that memory can freely intermix with any other application-allocated object. Also, if an application that dynamically links to MFC fails, the protection of the operating system maintains the integrity of any other MFC application sharing the DLL.

Similarly other global MFC states, like the current executable file to load resources from, are also shared between the client application and all MFC extension DLLs as well as MFCx0.dll itself.

Sharing Resources and Classes

Exporting resources is done through a resource list. Each application contains a singly linked list of **CDynLinkLibrary** objects. When looking for a resource, most of the standard MFC implementations that load resources look first at the current resource module (`AfxGetResourceHandle`) and if the resource is not found walk the list of **CDynLinkLibrary** objects attempting to load the requested resource.

Walking the list has the disadvantages that it is slightly slower and requires managing resource ID ranges. It has the advantage that a client application that links to several MFC extension DLLs can use any DLL-provided resource without having to specify the DLL instance handle. `AfxFindResourceHandle` is an API used for walking the resource list to look for a given match. It takes the name and type of a resource and returns the resource handle where it was first found (or NULL).

If you do not want to walk the list and only load resources from a specific place, use the functions `AfxGetResourceHandle` and `AfxSetResourceHandle` to save the old handle and set the new handle. Be sure to restore the old resource handle before you return to the client application. For an example of using this approach to explicitly load a menu, see Testdll2 .cpp in the MFC sample DLLHUSK.

Dynamic creation of MFC objects given an MFC name is similar. The MFC object deserialization mechanism needs to have all of the `CRuntimeClass` objects registered so that it can reconstruct by dynamically creating C++ objects of the required type based on what was stored earlier.

In the case of the MFC sample DLLHUSK, the list looks something like:

```
head ->   DLLHUSK.EXE    - or -   DLLHUSK.EXE
               |                       |
          TESTDLL2.DLL            TESTDLL2.DLL
               |                       |
          TESTDLL1.DLL            TESTDLL1.DLL
               |                       |
           MFCOxxD.DLL                 |
               |                       |
           MFCDxxD.DLL                 |
               |                       |
            MFCxxD.DLL              MFCxx.DLL
```

where *xx* is the version number; for example, 42 represents version 4.2.

The MFCxx.dll is usually last on the resource and class list. MFCxx.dll includes all of the standard MFC resources, including prompt strings for all of the standard command IDs. Placing it at the end of the list allows DLLs and the client application itself not to have their own copy of the standard MFC resources, but to rely on the shared resources in the MFCxx.dll instead.

Merging the resources and class names of all DLLs into the client application's name space has the disadvantage of requiring you to be careful with what IDs or names you pick.

The DLLHUSK sample manages the shared resource name space by using multiple header files.

If your MFC extension DLL needs to maintain extra data for each application, you can derive a new class from **CDynLinkLibrary** and create it in `DllMain`. When running, the DLL can check the current application's list of **CDynLinkLibrary** objects to find the one for that particular MFC extension DLL.

What do you want to do?

- Initialize an MFC extension DLL

What do you want to know more about?

- Tips on using shared resource files
- DLL Version of MFC

- Regular MFC DLLs statically linked to MFC
- Regular MFC DLLs dynamically linked to MFC
- Using Database, OLE, and Sockets MFC extension DLLs in regular MFC DLLs

See also

Create C/C++ DLLs in Visual Studio

Using Database, OLE, and Sockets MFC extension DLLs in regular MFC DLLs

Article • 08/03/2021

When using an MFC extension DLL from a regular MFC DLL, if the MFC extension DLL isn't wired into the `CDynLinkLibrary` object chain of the regular MFC DLL, you might run into one or more related problems. Because the debug versions of the MFC Database, OLE, and Sockets support DLLs are implemented as MFC extension DLLs, you might see similar problems if you're using these MFC features, even if you're not explicitly using any of your own MFC extension DLLs. Some symptoms are:

- When attempting to deserialize an object of a type of class defined in the MFC extension DLL, the message "Warning: Cannot load CYourClass from archive. Class not defined." appears in the TRACE debug window and the object fails to serialize.
- An exception indicating bad class might be thrown.
- Resources stored in the MFC extension DLL fail to load because `AfxFindResourceHandle` returns `NULL` or an incorrect resource handle.
- `DllGetClassObject`, `DllCanUnloadNow`, and the `UpdateRegistry`, `Revoke`, `RevokeAll`, and `RegisterAll` member functions of `COleObjectFactory` fail to locate a class factory defined in the MFC extension DLL.
- `AfxDoForAllClasses` doesn't work for any classes in the MFC extension DLL.
- Standard MFC database, sockets, or OLE resources fail to load. For example, `AfxLoadString(AFX_IDP_SQL_CONNECT_FAIL)` returns an empty string, even when the regular MFC DLL is properly using the MFC Database classes.

The solution to these problems is to create and export an initialization function in the MFC extension DLL that creates a `CDynLinkLibrary` object. Call this initialization function exactly once from each regular MFC DLL that uses the MFC extension DLL.

MFC OLE, MFC Database (or DAO), or MFC Sockets Support

If you're using any MFC OLE, MFC Database (or DAO), or MFC Sockets support in your regular MFC DLL, respectively, the MFC debug MFC extension DLLs *`MFCOxxD.dll`*,

MFCDxxD.dll, and *MFCNxxD.dll* (where *xx* is the version number) are linked automatically. Call a predefined initialization function for each of the DLLs that you're using:

- For database support, add a call to `AfxDbInitModule` to your regular MFC DLL in its `CWinApp::InitInstance` function. Make sure this call occurs before any base-class call or any added code that accesses the *MFCDxxD.dll*. This function takes no parameters and returns `void`.

- For OLE support, add a call to `AfxOleInitModule` to your regular MFC DLL it its `CWinApp::InitInstance` function. The `COleControlModule::InitInstance` function calls `AfxOleInitModule` already, so if you're building an OLE control and use `COleControlModule`, you shouldn't add this call to `AfxOleInitModule`.

- For Sockets support, add a call to `AfxNetInitModule` to your regular MFC DLL in `CWinApp::InitInstance`.

Release builds of MFC DLLs and applications don't use separate DLLs for database, sockets, or OLE support. However, it's safe to call these initialization functions in release mode.

CDynLinkLibrary Objects

During each operation mentioned at the beginning of this article, MFC needs to search for a particular value or object. For example, during deserialization, MFC needs to search through all the currently available run-time classes to match objects in the archive with their proper run-time class.

As a part of these searches, MFC scans through all the MFC extension DLLs in use by walking a chain of `CDynLinkLibrary` objects. `CDynLinkLibrary` objects attach automatically to a chain during their construction and are created by each MFC extension DLL in turn during initialization. Every module (application or regular MFC DLL) has its own chain of `CDynLinkLibrary` objects.

For an MFC extension DLL to get wired into a `CDynLinkLibrary` chain, it must create a `CDynLinkLibrary` object in the context of every module that uses the MFC extension DLL. To use an MFC extension DLL in regular MFC DLLs, the extension DLL must provide an exported initialization function that creates a `CDynLinkLibrary` object. Every regular MFC DLL that uses the MFC extension DLL must call the exported initialization function.

If you'll only use an MFC extension DLL from an MFC application, and never from a regular MFC DLL, then it's sufficient to create the `CDynLinkLibrary` object in the MFC

extension DLL `DllMain` function. It's what the MFC DLL Wizard MFC extension DLL code does. When loading an MFC extension DLL implicitly, `DllMain` loads and executes before the application ever starts. Any `CDynLinkLibrary` creations are wired into a default chain that the MFC DLL reserves for an MFC application.

It's a bad idea to have multiple `CDynLinkLibrary` objects from one MFC extension DLL in any one chain. It's especially true if the MFC extension DLL may be dynamically unloaded from memory. Don't call the initialization function more than once from any one module.

Sample Code

This sample code assumes that the regular MFC DLL implicitly links to the MFC extension DLL. To link implicitly, link to the import library (LIB file) of the MFC extension DLL when you build the regular MFC DLL.

The following lines should be in the source of the MFC extension DLL:

C++

```
// YourExtDLL.cpp:

// standard MFC extension DLL routines
#include "afxdllx.h"

static AFX_EXTENSION_MODULE extensionDLL;

extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        // MFC extension DLL one-time initialization
        if (!AfxInitExtensionModule(extensionDLL, hInstance))
           return 0;
    }
    return 1;   // ok
}

// Exported DLL initialization is run in context of
// application or regular MFC DLL
extern "C" void WINAPI InitYourExtDLL()
{
    // create a new CDynLinkLibrary for this app
    new CDynLinkLibrary(extensionDLL);

    // add other initialization here
}
```

Be sure to export the **InitYourExtDLL** function. You can use `__declspec(dllexport)`, or export it in the DEF file for your DLL, as shown here:

def

```
// YourExtDLL.Def:
LIBRARY      YOUREXTDLL
CODE         PRELOAD MOVEABLE DISCARDABLE
DATA         PRELOAD SINGLE
EXPORTS
    InitYourExtDLL
```

Add a call to the `InitInstance` member of the `CWinApp`-derived object in each regular MFC DLL using the MFC extension DLL:

C++

```
// YourRegularDLL.cpp:

class CYourRegularDLL : public CWinApp
{
public:
    virtual BOOL InitInstance(); // Initialization
    virtual int ExitInstance();  // Termination

    // nothing special for the constructor
    CYourRegularDLL(LPCTSTR pszAppName) : CWinApp(pszAppName) { }
};

BOOL CYourRegularDLL::InitInstance()
{
    // any DLL initialization goes here
    TRACE0("YOUR regular MFC DLL initializing\n");

    // wire any MFC extension DLLs into CDynLinkLibrary chain
    InitYourExtDLL();

    return TRUE;
}
```

What do you want to do?

- Initialize an MFC extension DLL
- Initialize regular MFC DLLs

What do you want to know more about?

- MFC extension DLLs
- Regular MFC DLLs Statically Linked to MFC
- Regular MFC DLLs Dynamically Linked to MFC
- Using MFC as Part of a DLL
- DLL Version of MFC

See also

MFC extension DLLs

Creating a resource-only DLL

Article • 08/03/2021

A resource-only DLL is a DLL that contains nothing but resources, such as icons, bitmaps, strings, and dialog boxes. Using a resource-only DLL is a good way to share the same set of resources among multiple programs. It's also a good way to provide an application with resources localized for multiple languages. For more information, see Localized resources in MFC applications: Satellite DLLs.

Create a resource-only DLL

To create a resource-only DLL, you create a new Windows DLL (non-MFC) project, and add your resources to the project:

1. Select **Windows Desktop Wizard** in the **New Project** dialog box and choose **Next**. In the **Configure your new project** page, enter the project and solution names, and choose **Create**.

2. In the **Windows Desktop Project** dialog box, select an **Application type** of **Dynamic Link Library**. Under **Additional options**, select **Empty project**. Choose **OK** to create your project.

3. Create a new resource script that contains the resources for the DLL (such as a string or a menu). Save the `.rc` file.

4. On the **Project** menu, select **Add Existing Item**, and then insert the new `.rc` file into the project.

5. Specify the /NOENTRY linker option. `/NOENTRY` prevents the linker from linking a reference to `_main` into the DLL; this option is required to create a resource-only DLL.

6. Build the DLL.

Use a resource-only DLL

The application that uses the resource-only DLL should call LoadLibraryEx or a related function to explicitly link to the DLL. To access the resources, call the generic functions `FindResource` and `LoadResource`, which work on any kind of resource. Or, call one of the following resource-specific functions:

- `FormatMessage`
- `LoadAccelerators`
- `LoadBitmap`
- `LoadCursor`
- `LoadIcon`
- `LoadMenu`
- `LoadString`

The application should call `FreeLibrary` when it's finished using the resources.

See also

Working with Resource Files
Create C/C++ DLLs in Visual Studio

Localized Resources in MFC Applications: Satellite DLLs

Article • 08/03/2021

MFC version 7.0 and later provides enhanced support for satellite DLLs, a feature that helps in creating applications localized for multiple languages. A satellite DLL is a resource-only DLL that contains an application's resources localized for a particular language. When the application begins executing, MFC automatically loads the localized resource most appropriate for the environment. For example, you could have an application with English language resources with two satellite DLLs, one containing a French translation of your resources and the other containing a German translation. When the application is run on an English language system, it uses the English resources. If run on a French system, it uses the French resources; if run on a German system, it uses the German resources.

To support localized resources in an MFC application, MFC attempts to load a satellite DLL containing resources localized to a specific language. Satellite DLLs are named *ApplicationNameXXX*.dll, where *ApplicationName* is the name of the .exe or .dll using MFC, and *XXX* is the three-letter code for the language of the resources (for example, 'ENU' or 'DEU').

MFC attempts to load the resource DLL for each of the following languages in order, stopping when it finds one:

1. The current user's default UI language, as returned from the GetUserDefaultUILanguage() Win32 API.

2. The current user's default UI language, without any specific sublanguage (that is, ENC [Canadian English] becomes ENU [U.S. English]).

3. The system's default UI language, as returned from the GetSystemDefaultUILanguage() API. On other platforms, this is the language of the OS itself.

4. The system's default UI language, without any specific sublanguage.

5. A fake language with the 3-letter code LOC.

If MFC does not find any satellite DLLs, it uses whatever resources are contained in the application itself.

As an example, suppose that an application LangExample.exe uses MFC and is running on a multiple user-interface system; the system UI language is ENU [U.S. English] and the current user's UI language is set to FRC [Canadian French]. MFC looks for the following DLLs in the following order:

1. LangExampleFRC.dll (user's UI language).
2. LangExampleFRA.dll (user's UI language without the sublanguage, in this example French (France).
3. LangExampleENU.dll (system's UI language).
4. LangExampleLOC.dll.

If none of these DLLs are found, MFC uses the resources in LangExample.exe.

See also

Create C/C++ DLLs in Visual Studio
TN057: Localization of MFC Components

Importing and Exporting

Article • 08/03/2021

You can import public symbols into an application or export functions from a DLL using two methods:

- Use a module definition (.def) file when building the DLL
- Use the keywords `__declspec(dllimport)` or `__declspec(dllexport)` in a function definition in the main application

Using a .def file

A module-definition (.def) file is a text file containing one or more module statements that describe various attributes of a DLL. If you do not use `__declspec(dllimport)` or `__declspec(dllexport)` to export a DLL's functions, the DLL requires a .def file.

You can use .def files to import into an application or to export from a DLL.

Using __declspec

You do not need to use `__declspec(dllimport)` for your code to compile correctly, but doing so allows the compiler to generate better code. The compiler is able to generate better code because it can determine whether a function exists in a DLL or not, which allows the compiler to produce code that skips a level of indirection that would normally be present in a function call that crossed a DLL boundary. However, you must use `__declspec(dllimport)` to import variables used in a DLL.

With the proper .def file EXPORTS section, `__declspec(dllexport)` is not required. `__declspec(dllexport)` was added to provide an easy way to export functions from an .exe or .dll file without using a .def file.

The Win32 Portable Executable format is designed to minimize the number of pages that must be touched to fix imports. To do this, it places all the import addresses for any program in one place called the Import Address Table. This allows the loader to modify only one or two pages when accessing these imports.

What do you want to do?

- Import into an Application

- Export from a DLL

See also

Create C/C++ DLLs in Visual Studio

Importing into an Application

Article • 08/03/2021

You can import functions into an application using two methods:

- Use the keywords `__declspec(dllimport)` in a function definition in the main application
- Use a module definition (.def) file along with `__declspec(dllimport)`

What do you want to do?

- Import into an Application Using __declspec(dllimport)
- Import Function Calls Using __declspec(dllimport)
- Import Data Using __declspec(dllimport)
- Import Using DEF Files

See also

Importing and Exporting

Import into an application using __declspec(dllimport)

Article • 08/03/2021

A program that uses public symbols defined by a DLL is said to import them. When you create header files for applications that use your DLLs to build with, use `__declspec(dllimport)` on the declarations of the public symbols. The keyword `__declspec(dllimport)` works whether you export with .def files or with the `__declspec(dllexport)` keyword.

To make your code more readable, define a macro for `__declspec(dllimport)` and then use the macro to declare each imported symbol:

```
#define DllImport   __declspec( dllimport )

DllImport int  j;
DllImport void func();
```

Using `__declspec(dllimport)` is optional on function declarations, but the compiler produces more efficient code if you use this keyword. However, you must use `__declspec(dllimport)` for the importing executable to access the DLL's public data symbols and objects. Note that the users of your DLL still need to link with an import library.

You can use the same header file for both the DLL and the client application. To do this, use a special preprocessor symbol that indicates whether you are building the DLL or building the client application. For example:

```
#ifdef _EXPORTING
   #define CLASS_DECLSPEC    __declspec(dllexport)
#else
   #define CLASS_DECLSPEC    __declspec(dllimport)
#endif

class CLASS_DECLSPEC CExampleA : public CObject
{ ... class definition ... };
```

What do you want to do?

- Initialize a DLL

What do you want to know more about?

- Importing and exporting inline functions
- Mutual imports

See also

Importing into an Application

Importing function calls using `__declspec(dllimport)`

Article • 08/03/2021

Annotating calls by using the `__declspec(dllimport)` can make them faster. `__declspec(dllimport)` is always required to access exported DLL data.

Import a function from a DLL

The following code example shows how to use `__declspec(dllimport)` to import function calls from a DLL into an application. Assume that `func1` is a function that's in a DLL separate from the executable file that contains the **main** function.

Without `__declspec(dllimport)`, given this code:

```C
int main(void)
{
   func1();
}
```

the compiler generates code that looks like this:

```asm
call func1
```

and the linker translates the call into something like this:

```asm
call 0x4000000         ; The address of 'func1'.
```

If `func1` exists in another DLL, the linker can't resolve this address directly because it has no way of knowing what the address of `func1` is. In 32-bit and 64-bit environments, the linker generates a thunk at a known address. In a 32-bit environment the thunk looks like:

```asm
```

```
0x40000000:    jmp DWORD PTR __imp_func1
```

Here `__imp_func1` is the address for the `func1` slot in the import address table of the executable file. All these addresses are known to the linker. The loader only has to update the executable file's import address table at load time for everything to work correctly.

That's why using **`__declspec(dllimport)`** is better: because the linker doesn't generate a thunk if it's not required. Thunks make the code larger (on RISC systems, it can be several instructions) and can degrade your cache performance. If you tell the compiler the function is in a DLL, it can generate an indirect call for you.

So now this code:

C

```
__declspec(dllimport) void func1(void);
int main(void)
{
   func1();
}
```

generates this instruction:

asm

```
call DWORD PTR __imp_func1
```

There's no thunk and no `jmp` instruction, so the code is smaller and faster. You can also get the same effect without **`__declspec(dllimport)`** by using whole program optimization. For more information, see /GL (Whole Program Optimization).

For function calls within a DLL, you don't want to have to use an indirect call. The linker already knows the function's address. It takes extra time and space to load and store the address of the function before an indirect call. A direct call is always faster and smaller. You only want to use **`__declspec(dllimport)`** when calling DLL functions from outside the DLL itself. Don't use **`__declspec(dllimport)`** on functions inside a DLL when building that DLL.

See also

Importing into an Application

Importing data using `__declspec(dllimport)`

Article • 08/03/2021

In the case of data, using `__declspec(dllimport)` is a convenience item that removes a layer of indirection. When you import data from a DLL, you still have to go through the import address table. Before `__declspec(dllimport)`, this meant you had to remember to do an extra level of indirection when accessing data exported from the DLL:

C

```
// project.h
// Define PROJECT_EXPORTS when building your DLL
#ifdef PROJECT_EXPORTS   // If accessing the data from inside the DLL
   ULONG ulDataInDll;

#else         // If accessing the data from outside the DLL
   ULONG *ulDataInDll;
#endif
```

You would then export the data in your .DEF file:

DEF

```
// project.def
LIBRARY project
EXPORTS
   ulDataInDll   CONSTANT
```

and access it outside the DLL:

C

```
if (*ulDataInDll == 0L)
{
   // Do stuff here
}
```

When you mark the data as `__declspec(dllimport)`, the compiler automatically generates the indirection code for you. You no longer have to worry about the steps above. As stated previously, do not use `__declspec(dllimport)` declaration on the data when building the DLL. Functions within the DLL do not use the import address table to access the data object; therefore, you will not have the extra level of indirection present.

To export the data automatically from the DLL, use this declaration:

C

```
// project.h
// Define PROJECT_EXPORTS when building your DLL
#ifdef PROJECT_EXPORTS   // If accessing the data from inside the DLL
   __declspec(dllexport) ULONG ulDataInDLL;

#else         // If accessing the data from outside the DLL
   __declspec(dllimport) ULONG ulDataInDLL;
#endif
```

See also

Importing into an Application

Importing Using DEF Files

Article • 08/03/2021

If you choose to use `__declspec(dllimport)` along with a .def file, you should change the .def file to use DATA in place of CONSTANT to reduce the likelihood that incorrect coding will cause a problem:

```
// project.def
LIBRARY project
EXPORTS
   ulDataInDll   DATA
```

The following table shows why.

Keyword	Emits in the import library	Exports
`CONSTANT`	`_imp_ulDataInDll`, `_ulDataInDll`	`_ulDataInDll`
`DATA`	`_imp_ulDataInDll`	`_ulDataInDll`

Using `__declspec(dllimport)` and CONSTANT lists both the `imp` version and the undecorated name in the .lib DLL import library that is created to allow explicit linking. Using `__declspec(dllimport)` and DATA lists just the `imp` version of the name.

If you use CONSTANT, either of the following code constructs can be used to access `ulDataInDll`:

```
__declspec(dllimport) ULONG ulDataInDll; /*prototype*/
if (ulDataInDll == 0L)   /*sample code fragment*/
```

-or-

```
ULONG *ulDataInDll;      /*prototype*/
if (*ulDataInDll == 0L)  /*sample code fragment*/
```

However, if you use DATA in your .def file, only code compiled with the following definition can access the variable `ulDataInDll`:

```
__declspec(dllimport) ULONG ulDataInDll;

if (ulDataInDll == 0L)   /*sample code fragment*/
```

Using CONSTANT is more risky because if you forget to use the extra level of indirection, you could potentially access the import address table's pointer to the variable — not the variable itself. This type of problem can often manifest as an access violation because the import address table is currently made read-only by the compiler and linker.

The current MSVC linker issues a warning if it sees CONSTANT in the .def file to account for this case. The only real reason to use CONSTANT is if you cannot recompile some object file where the header file did not list `__declspec(dllimport)` on the prototype.

See also

Importing into an Application

Exporting from a DLL

Article • 08/03/2021

A DLL file has a layout very similar to an .exe file, with one important difference — a DLL file contains an exports table. The exports table contains the name of every function that the DLL exports to other executables. These functions are the entry points into the DLL; only the functions in the exports table can be accessed by other executables. Any other functions in the DLL are private to the DLL. The exports table of a DLL can be viewed by using the DUMPBIN tool with the /EXPORTS option.

You can export functions from a DLL using two methods:

- Create a module definition (.def) file and use the .def file when building the DLL. Use this approach if you want to export functions from your DLL by ordinal rather than by name.
- Use the keyword `__declspec(dllexport)` in the function's definition.

When exporting functions with either method, make sure to use the __stdcall calling convention.

What do you want to do?

- Export from a DLL using .def files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Export C functions for use in C or C++-language executables
- Export functions from a DLL by ordinal rather than by name
- Determine which exporting method to use
- Link an executable to a DLL
- Initialize a DLL

What do you want to know more about?

- Importing into an application
- Importing and exporting inline functions
- Mutual imports

See also

Importing and Exporting

Exporting from a DLL Using DEF Files

Article • 08/03/2021

A module-definition or DEF file (*.def) is a text file containing one or more module statements that describe various attributes of a DLL. If you are not using the `__declspec(dllexport)` keyword to export the DLL's functions, the DLL requires a DEF file.

A minimal DEF file must contain the following module-definition statements:

- The first statement in the file must be the LIBRARY statement. This statement identifies the DEF file as belonging to a DLL. The LIBRARY statement is followed by the name of the DLL. The linker places this name in the DLL's import library.
- The EXPORTS statement lists the names and, optionally, the ordinal values of the functions exported by the DLL. You assign the function an ordinal value by following the function's name with an at sign (@) and a number. When you specify ordinal values, they must be in the range 1 through N, where N is the number of functions exported by the DLL. If you want to export functions by ordinal, see Exporting Functions from a DLL by Ordinal Rather Than by Name as well as this topic.

For example, a DLL that contains the code to implement a binary search tree might look like the following:

```
LIBRARY   BTREE
EXPORTS
   Insert   @1
   Delete   @2
   Member   @3
   Min   @4
```

If you use the MFC DLL Wizard to create an MFC DLL, the wizard creates a skeleton DEF file for you and automatically adds it to your project. Add the names of the functions to be exported to this file. For non-MFC DLLs, create the DEF file yourself and add it to your project. Then go to **Project** > **Properties** > **Linker** > **Input** > **Module Definition File** and enter the name of the DEF file. Repeat this step for each configuration and platform, or do it all at once by selecting **Configuration** = **All Configurations**, and **Platform** = **All Platforms**.

If you are exporting functions in a C++ file, you have to either place the decorated names in the DEF file or define your exported functions with standard C linkage by using extern "C". If you need to place the decorated names in the DEF file, you can obtain them by using the DUMPBIN tool or by using the linker /MAP option. Note that the decorated names produced by the compiler are compiler specific. If you place the decorated names produced by the Microsoft C++ compiler (MSVC) into a DEF file, applications that link to your DLL must also be built using the same version of MSVC so that the decorated names in the calling application match the exported names in the DLL's DEF file.

> **Note**
>
> A DLL built with Visual Studio 2015 can be consumed by applications built with Visual Studio 2017 or Visual Studio 2019.

If you are building an extension DLL, and exporting using a DEF file, place the following code at the beginning and end of your header files that contain the exported classes:

```
#undef AFX_DATA
#define AFX_DATA AFX_EXT_DATA
// <body of your header file>
#undef AFX_DATA
#define AFX_DATA
```

These lines ensure that MFC variables that are used internally or that are added to your classes are exported (or imported) from your MFC extension DLL. For example, when deriving a class using `DECLARE_DYNAMIC`, the macro expands to add a `CRuntimeClass` member variable to your class. Leaving out these four lines might cause your DLL to compile or link incorrectly or cause an error when the client application links to the DLL.

When building the DLL, the linker uses the DEF file to create an export (.exp) file and an import library (.lib) file. The linker then uses the export file to build the DLL file. Executables that implicitly link to the DLL link to the import library when they are built.

Note that MFC itself uses DEF files to export functions and classes from the MFCx0.dll.

What do you want to do?

- Export from a DLL using __declspec(dllexport)

- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Export C functions for use in C or C++-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- .def files
- Rules for module-definition statements
- Decorated names
- Importing and exporting inline functions
- Mutual imports

See also

Exporting from a DLL

Exporting from a DLL Using __declspec(dllexport)

Article • 08/03/2021

You can export data, functions, classes, or class member functions from a DLL using the `__declspec(dllexport)` keyword. `__declspec(dllexport)` adds the export directive to the object file so you do not need to use a .def file.

This convenience is most apparent when trying to export decorated C++ function names. Because there is no standard specification for name decoration, the name of an exported function might change between compiler versions. If you use `__declspec(dllexport)`, recompiling the DLL and dependent .exe files is necessary only to account for any naming convention changes.

Many export directives, such as ordinals, NONAME, and PRIVATE, can be made only in a .def file, and there is no way to specify these attributes without a .def file. However, using `__declspec(dllexport)` in addition to using a .def file does not cause build errors.

To export functions, the `__declspec(dllexport)` keyword must appear to the left of the calling-convention keyword, if a keyword is specified. For example:

```
__declspec(dllexport) void __cdecl Function1(void);
```

To export all of the public data members and member functions in a class, the keyword must appear to the left of the class name as follows:

```
class __declspec(dllexport) CExampleExport : public CObject
{ ... class definition ... };
```

> **Note**
>
> `__declspec(dllexport)` cannot be applied to a function with the `__clrcall` calling convention.

When building your DLL, you typically create a header file that contains the function prototypes and/or classes you are exporting and add `__declspec(dllexport)` to the

declarations in the header file. To make your code more readable, define a macro for `__declspec(dllexport)` and use the macro with each symbol you are exporting:

```
#define DllExport   __declspec( dllexport )
```

`__declspec(dllexport)` stores function names in the DLL's export table. If you want to optimize the table's size, see Exporting Functions from a DLL by Ordinal Rather Than by Name.

What do you want to do?

- Export from a DLL using .def files
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Export C functions for use in C or C++-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- The __declspec keyword
- Importing and exporting inline functions
- Mutual imports

See also

Exporting from a DLL

Exporting and Importing Using AFX_EXT_CLASS

Article • 08/03/2021

MFC extension DLLs use the macro **AFX_EXT_CLASS** to export classes; the executables that link to the MFC extension DLL use the macro to import classes. With the **AFX_EXT_CLASS** macro, the same header files that are used to build the MFC extension DLL can be used with the executables that link to the DLL.

In the header file for your DLL, add the **AFX_EXT_CLASS** keyword to the declaration of your class as follows:

C++

```
class AFX_EXT_CLASS CMyClass : public CDocument
{
// <body of class>
};
```

This macro is defined by MFC as `__declspec(dllexport)` when the preprocessor symbols `_AFXDLL` and `_AFXEXT` are defined. But the macro is defined as `__declspec(dllimport)` when `_AFXDLL` is defined and `_AFXEXT` is not defined. When defined, the preprocessor symbol `_AFXDLL` indicates that the shared version of MFC is being used by the target executable (either a DLL or an application). When both `_AFXDLL` and `_AFXEXT` are defined, this indicates that the target executable is an MFC extension DLL.

Because `AFX_EXT_CLASS` is defined as `__declspec(dllexport)` when exporting from an MFC extension DLL, you can export entire classes without placing the decorated names for all of that class's symbols in the .def file.

Although you can avoid creating a .def file and all of the decorated names for the class with this method, creating a .def file is more efficient because the names can be exported by ordinal. To use the .def file method of exporting, place the following code at the beginning and end of your header file:

C++

```
#undef AFX_DATA
#define AFX_DATA AFX_EXT_DATA
// <body of your header file>
#undef AFX_DATA
#define AFX_DATA
```

⊗ **Caution**

Be careful when exporting inline functions, because they can create the possibility of version conflicts. An inline function gets expanded into the application code; therefore, if you later rewrite the function, it does not get updated unless the application itself is recompiled. Normally, DLL functions can be updated without rebuilding the applications that use them.

Exporting Individual Members in a Class

Sometimes you might want to export individual members of your class. For example, if you are exporting a `CDialog`-derived class, you might only need to export the constructor and the `DoModal` call. You can use `AFX_EXT_CLASS` on the individual members you need to export.

For example:

C++

```
class CExampleDialog : public CDialog
{
public:
   AFX_EXT_CLASS CExampleDialog();
   AFX_EXT_CLASS int DoModal();
   ...
   // rest of class definition
   ...
};
```

Because you are no longer exporting all members of the class, you may run into an additional problem because of the way that MFC macros work. Several of MFC's helper macros actually declare or define data members. Therefore, these data members must also be exported from your DLL.

For example, the `DECLARE_DYNAMIC` macro is defined as follows when building an MFC extension DLL:

C++

```
#define DECLARE_DYNAMIC(class_name) \
protected: \
   static CRuntimeClass* PASCAL _GetBaseClass(); \
public: \
```

```
   static AFX_DATA CRuntimeClass class##class_name; \
   virtual CRuntimeClass* GetRuntimeClass() const; \
```

The line that begins with static `AFX_DATA` is declaring a static object inside of your class. To export this class correctly and access the run-time information from a client executable, you must export this static object. Because the static object is declared with the modifier `AFX_DATA`, you only need to define `AFX_DATA` to be `__declspec(dllexport)` when building your DLL and define it as `__declspec(dllimport)` when building your client executable. Because `AFX_EXT_CLASS` is already defined in this way, you just need to redefine `AFX_DATA` to be the same as `AFX_EXT_CLASS` around your class definition.

For example:

C++

```
#undef  AFX_DATA
#define AFX_DATA AFX_EXT_CLASS

class CExampleView : public CView
{
   DECLARE_DYNAMIC()
   // ... class definition ...
};

#undef  AFX_DATA
#define AFX_DATA
```

Because MFC always uses the `AFX_DATA` symbol on data items it defines within its macros, this technique works for all such scenarios. For example, it works for `DECLARE_MESSAGE_MAP`.

> **Note**
>
> If you are exporting the entire class rather than selected members of the class, static data members are automatically exported.

What do you want to do?

- Export from a DLL using .def files
- Export from a DLL using __declspec(dllexport)
- Export C++ functions for use in C-language executables

- Export C functions for use in C or C++-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- Decorated names
- Importing and exporting inline functions
- Mutual imports

See also

Exporting from a DLL

Exporting C++ Functions for Use in C-Language Executables

Article • 08/03/2021

If you have functions in a DLL written in C++ that you want to access from a C-language module, you should declare these functions with C linkage instead of C++ linkage. Unless otherwise specified, the C++ compiler uses C++ type-safe naming (also known as name decoration) and C++ calling conventions, which can be difficult to call from C.

To specify C linkage, specify `extern "C"` for your function declarations. For example:

```
extern "C" __declspec( dllexport ) int MyFunc(long parm1);
```

What do you want to do?

- Export from a DLL using .def files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C functions for use in C or C++-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- Decorated names
- Using extern to Specify Linkage

See also

Exporting from a DLL

Export C functions for use in C or C++ language executables

Article • 05/25/2022

If you have functions in a DLL written in C, you can use a preprocessor macro to make them easy to access from both C language and C++ language code. The `__cplusplus` preprocessor macro indicates which language is being compiled. You may use it to declare the functions with C linkage when called from C++ language code. If you use this technique and provide header files for your DLL, these functions can be used by C and C++ users with no change.

The following code shows a header file that both C and C++ client applications can use:

h

```
// MyCFuncs.h
#ifdef __cplusplus
extern "C" {  // only need to export C interface if
              // used by C++ source code
#endif

__declspec( dllimport ) void MyCFunc();
__declspec( dllimport ) void AnotherCFunc();

#ifdef __cplusplus
}
#endif
```

Sometimes you may need to link C functions to your C++ executable, but the function declaration header files haven't used the above technique. You can still call the functions from C++. In the C++ source file, wrap the `#include` directive to prevent the compiler from decorating the C function names:

C++

```
extern "C" {
#include "MyCHeader.h"
}
```

What do you want to do?

- Export from a DLL using .def files

- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- Decorated names
- Using extern to specify linkage

See also

Exporting from a DLL

Determine Which Exporting Method to Use

Article • 08/03/2021

You can export functions in either of two ways—a .def file or the `__declspec(dllexport)` keyword. To help you decide which way is better for your DLL, consider these questions:

- Do you plan to export more functions later?
- Is your DLL used only by applications that you can rebuild, or is it used by applications that you cannot rebuild—for example, applications that are created by third parties?

Pros and Cons of Using .def Files

Exporting functions in a .def file gives you control over the export ordinals. When you add an exported function to your DLL, you can assign it a higher ordinal value than any other exported function. When you do this, applications that use implicit linking do not have to relink with the import library that contains the new function. This is very convenient if you are designing a DLL for use by many applications because you can add new functionality and also ensure that it continues to work correctly with the applications that already rely on it. For example, the MFC DLLs are built by using .def files.

Another advantage to using a .def file is that you can use the `NONAME` attribute to export a function. This puts only the ordinal in the exports table in the DLL. For DLLs that have a large number of exported functions, using the `NONAME` attribute can reduce the size of the DLL file. For information about how to write a module definition statement, see Rules for Module-Definition Statements. For information about ordinal export, see Exporting Functions from a DLL by Ordinal Rather Than by Name.

A disadvantage of using a .def file is that if you are exporting functions in a C++ file, you either have to put the decorated names in the .def file or define the exported functions by using extern "C" to avoid the name decoration that's done by the MSVC compiler.

If you put the decorated names in the .def file, you can obtain them by using the DUMPBIN tool or by using the linker /MAP option. The decorated names that are produced by the compiler are compiler-specific; therefore, if you put the decorated names that are produced by the compiler into a .def file, the applications that link to the

DLL must also be built by using the same version of the compiler so that the decorated names in the calling application match the exported names in the .def file of the DLL.

Pros and Cons of Using __declspec(dllexport)

Using `__declspec(dllexport)` is convenient because you do not have to worry about maintaining a .def file and obtaining the decorated names of the exported functions. However, the usefulness of this way of exporting is limited by the number of linked applications that you are willing to rebuild. If you rebuild the DLL with new exports, you also have to rebuild the applications because the decorated names for exported C++ functions might change if you use a different version of the compiler to rebuild it.

What do you want to do?

- Export from a DLL using .DEF files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Export C functions for use in C or C++-language executables
- Import into an application using __declspec(dllimport)
- Initialize a DLL

What do you want to know more about?

- Importing and exporting inline functions
- Mutual imports
- Decorated names

See also

Exporting from a DLL

Exporting Functions from a DLL by Ordinal Rather Than by Name

Article • 08/03/2021

The simplest way to export functions from your DLL is to export them by name. This is what happens when you use `__declspec(dllexport)`, for example. But you can instead export functions by ordinal. With this technique, you must use a .def file instead of `__declspec(dllexport)`. To specify a function's ordinal value, append its ordinal to the function name in the .def file. For information about specifying ordinals, see Exporting from a DLL Using .def Files.

> **Tip**
>
> If you want to optimize your DLL's file size, use the **NONAME** attribute on each exported function. With the **NONAME** attribute, the ordinals are stored in the DLL's export table rather than the function names. This can be a considerable savings if you are exporting many functions.

What do you want to do?

- Use a .def file so I can export by ordinal
- Use __declspec(dllexport)

See also

Exporting from a DLL

Mutual Imports

Article • 08/03/2021

Exporting or importing to another executable file presents complications when the imports are mutual (or circular). For example, two DLLs import symbols from each other, similar to mutually recursive functions.

The problem with mutually importing executable files (usually DLLs) is that neither can be built without building the other first. Each build process requires, as input, an import library produced by the other build process.

The solution is to use the LIB utility with the /DEF option, which produces an import library without building the executable file. Using this utility, you can build all the import libraries you need, no matter how many DLLs are involved or how complicated the dependencies are.

The general solution for handling mutual imports is:

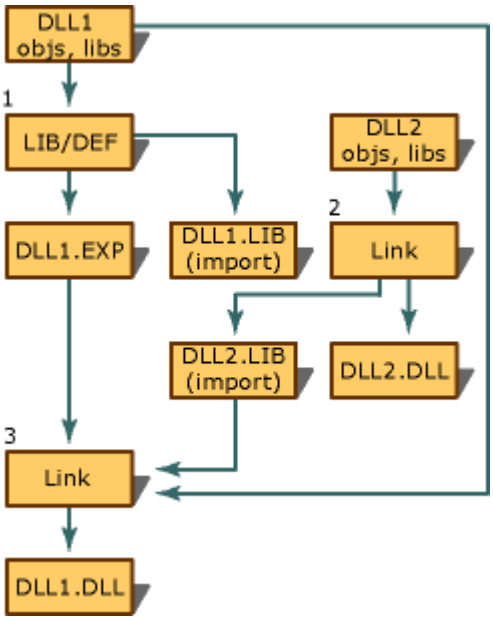
1. Take each DLL in turn. (Any order is feasible, although some orders are more optimal.) If all the needed import libraries exist and are current, run LINK to build the executable file (DLL). This produces an import library. Otherwise, run LIB to produce an import library.

 Running LIB with the /DEF option produces an additional file with an .EXP extension. The .EXP file must be used later to build the executable file.

2. After using either LINK or LIB to build all of the import libraries, go back and run LINK to build any executable files that were not built in the previous step. Note that the corresponding .exp file must be specified on the LINK line.

 If you had run the LIB utility earlier to produce an import library for DLL1, LIB would have produced the file DLL1.exp as well. You must use DLL1.exp as input to LINK when building DLL1.dlll.

The following illustration shows a solution for two mutually importing DLLs, DLL1 and DLL2. Step 1 is to run LIB, with the /DEF option set, on DLL1. Step 1 produces DLL1.lib, an import library, and DLL1.exp. In step 2, the import library is used to build DLL2, which in turn produces an import library for DLL2's symbols. Step 3 builds DLL1, by using DLL1.exp and DLL2.lib as input. Note that an .exp file for DLL2 is not necessary because LIB was not used to build DLL2's import library.



Linking Two DLLs with Mutual Imports

Limitations of _AFXEXT

You can use the `_AFXEXT` preprocessor symbol for your MFC extension DLLs as long as you do not have multiple layers of MFC extension DLLs. If you have MFC extension DLLs that call or derive from classes in your own MFC extension DLLs, which then derive from the MFC classes, you must use your own preprocessor symbol to avoid ambiguity.

The problem is that in Win32, you must explicitly declare any data as `__declspec(dllexport)` if it is to be exported from a DLL, and `__declspec(dllimport)` if it is to be imported from a DLL. When you define `_AFXEXT`, the MFC headers make sure that **AFX_EXT_CLASS** is defined correctly.

When you have multiple layers, one symbol such as **AFX_EXT_CLASS** is not sufficient, because an MFC extension DLL might be exporting new classes as well as importing other classes from another MFC extension DLL. To solve this problem, use a special preprocessor symbol that indicates that you are building the DLL itself versus using the DLL. For example, imagine two MFC extension DLLs, A.dll and B.dll. They each export some classes in A.h and B.h, respectively. B.dll uses the classes from A.dll. The header files would look something like this:

```
/* A.H */
#ifdef A_IMPL
   #define CLASS_DECL_A   __declspec(dllexport)
#else
   #define CLASS_DECL_A   __declspec(dllimport)
#endif
```

```
class CLASS_DECL_A CExampleA : public CObject
{ ... class definition ... };

// B.H
#ifdef B_IMPL
   #define CLASS_DECL_B   __declspec(dllexport)
#else
   #define CLASS_DECL_B   __declspec(dllimport)
#endif

class CLASS_DECL_B CExampleB : public CExampleA
{ ... class definition ... };
...
```

When A.dll is built, it is built with `/D A_IMPL` and when B.dll is built, it is built with `/D B_IMPL`. By using separate symbols for each DLL, `CExampleB` is exported and `CExampleA` is imported when building B.dll. `CExampleA` is exported when building A.dll and imported when used by B.dll (or some other client).

This type of layering cannot be done when using the built-in **AFX_EXT_CLASS** and `_AFXEXT` preprocessor symbols. The technique described above solves this problem in a manner not unlike the mechanism MFC itself uses when building its Active technologies, Database, and Network MFC extension DLLs.

Not Exporting the Entire Class

When you are not exporting an entire class, you have to ensure that the necessary data items created by the MFC macros are exported correctly. This can be done by redefining `AFX_DATA` to your specific class's macro. This should be done any time you are not exporting the entire class.

For example:

```
/* A.H */
#ifdef A_IMPL
   #define CLASS_DECL_A  _declspec(dllexport)
#else
   #define CLASS_DECL_A  _declspec(dllimport)
#endif

#undef  AFX_DATA
#define AFX_DATA CLASS_DECL_A

class CExampleA : public CObject
{
```

```
   DECLARE_DYNAMIC()
   CLASS_DECL_A int SomeFunction();
   //... class definition ...
};

#undef AFX_DATA
#define AFX_DATA
```

What do you want to do?

- Export from a DLL
- Export from a DLL using .DEF files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)

What do you want to know more about?

- The LIB utility and the /DEF option

See also

Importing and Exporting

Importing and exporting inline functions

Article • 08/03/2021

Imported functions can be defined as inline. The effect is roughly the same as defining a standard function inline; calls to the function are expanded into inline code, much like a macro. This is principally useful as a way of supporting C++ classes in a DLL that might inline some of their member functions for efficiency.

One feature of an imported inline function is that you can take its address in C++. The compiler returns the address of the copy of the inline function residing in the DLL. Another feature of imported inline functions is that you can initialize static local data of the imported function, unlike global imported data.

> **⊗ Caution**
>
> You should exercise care when providing imported inline functions because they can create the possibility of version conflicts. An inline function gets expanded into the application code; therefore, if you later rewrite the function, it does not get updated unless the application itself is recompiled. (Normally, DLL functions can be updated without rebuilding the applications that use them.)

What do you want to do?

- Export from a DLL
- Export from a DLL using .DEF files
- Export from a DLL using __declspec(dllexport)
- Export and import using AFX_EXT_CLASS
- Export C++ functions for use in C-language executables
- Determine which exporting method to use
- Import into an application using __declspec(dllimport)

See also

Active Technology and DLLs

Article • 08/03/2021

Active technology allows object servers to be completely implemented inside a DLL. This type of server is called an in-process server. MFC does not completely support in-process servers for all the features of visual editing, mainly because Active technology does not provide a way for a server to hook into the container's main message loop. MFC requires access to the container application's message loop to handle accelerator keys and idle-time processing.

If you are writing an Automation server and your server has no user interface, you can make your server an in-process server and put it completely into a DLL.

What do you want to know more about?

- Automation Servers

See also

Create C/C++ DLLs in Visual Studio

Automation in a DLL

Article • 08/03/2021

When you choose the Automation option in the MFC DLL Wizard, the wizard provides you with the following:

- A starter object description language (.ODL) file
- An include directive in the STDAFX.h file for Afxole.h
- An implementation of the `DllGetClassObject` function, which calls the **AfxDllGetClassObject** function
- An implementation of the `DllCanUnloadNow` function, which calls the **AfxDllCanUnloadNow** function
- An implementation of the `DllRegisterServer` function, which calls the COleObjectFactory::UpdateRegistryAll function

What do you want to know more about?

- Automation Servers

See also

Create C/C++ DLLs in Visual Studio

Calling DLL Functions from Visual Basic Applications

Article • 08/03/2021

For Visual Basic applications (or applications in other languages such as Pascal or Fortran) to call functions in a C/C++ DLL, the functions must be exported using the correct calling convention without any name decoration done by the compiler

`__stdcall` creates the correct calling convention for the function (the called function cleans up the stack and parameters are passed from right to left) but decorates the function name differently. So, when `__declspec(dllexport)` is used on an exported function in a DLL, the decorated name is exported.

The `__stdcall` name decoration prefixes the symbol name with an underscore (_) and appends the symbol with an at sign (@) character followed by the number of bytes in the argument list (the required stack space). As a result, the function when declared as:

```
C

int __stdcall func (int a, double b)
```

is decorated as `_func@12` in the output.

The C calling convention (`__cdecl`) decorates the name as `_func`.

To get the decorated name, use /MAP. Use of `__declspec(dllexport)` does the following:

- If the function is exported with the C calling convention (`__cdecl`), it strips the leading underscore (_) when the name is exported.
- If the function being exported does not use the C calling convention (for example, `__stdcall`), it exports the decorated name.

Because there is no way to override where the stack cleanup occurs, you must use `__stdcall`. To undecorate names with `__stdcall`, you must specify them by using aliases in the EXPORTS section of the .def file. This is shown as follows for the following function declaration:

```
C
```

```
int  __stdcall MyFunc (int a, double b);
void __stdcall InitCode (void);
```

In the .DEF file:

```
EXPORTS
   MYFUNC=_MyFunc@12
   INITCODE=_InitCode@0
```

For DLLs to be called by programs written in Visual Basic, the alias technique shown in this topic is needed in the .def file. If the alias is done in the Visual Basic program, use of aliasing in the .def file is not necessary. It can be done in the Visual Basic program by adding an alias clause to the Declare statement.

What do you want to know more about?

- Exporting from a DLL
- Exporting from a DLL using .DEF files
- Exporting from a DLL using __declspec(dllexport)
- Exporting C++ functions for use in C-language executables
- Determine which exporting method to use
- Decorated names

See also

Create C/C++ DLLs in Visual Studio

Building C/C++ Isolated Applications and Side-by-side Assemblies

Article • 08/03/2021

Visual Studio supports a deployment model for Windows client applications based on the idea of isolated applications and side-by-side assemblies. By default, Visual Studio builds all native C/C++ applications as isolated applications that use manifests to describe their dependencies on Visual C++ libraries.

Building C/C++ programs as isolated applications presents a range of advantages. For example, an isolated application is unaffected when other C/C++ applications install or uninstall Visual C++ libraries. Visual C++ libraries used by isolated applications may still be redistributed in either the application's local folder, or by installation to the native assembly cache (WinSxS); however, servicing of Visual C++ libraries for already deployed applications can be simplified by using a publisher configuration file. The isolated application deployment model makes it easier to ensure that C/C++ applications that are running on a specific computer use the most recent version of Visual C++ libraries, while still leaving open the possibility for system administrators and application authors to control explicit version binding of applications to their dependent DLLs.

This section discusses how you can build your C/C++ application as an isolated application and ensure that it binds to Visual C++ libraries using a manifest. The information in this section primarily applies to native, or unmanaged, C++ applications. For information about deploying native C++ applications built with Visual Studio, see Redistributing Visual C++ Files.

In This Section

Concepts of Isolated Applications and Side-by-side Assemblies

Building C/C++ Isolated Applications

Building C/C++ Side-by-side Assemblies

How to: Build Registration-Free COM Components

How to: Build Isolated Applications to Consume COM Components

Understanding Manifest Generation for C/C++ Programs

Related Sections

Isolated Applications and Side-by-side Assemblies

Deploying Desktop Applications

Concepts of Isolated Applications and Side-by-side Assemblies

Article • 08/03/2021

An application is considered an isolated application if all of its components are side-by-side assemblies. A side-by-side assembly is a collection of resources—a group of DLLs, windows classes, COM servers, type libraries, or interfaces—that are deployed together and made available for an application to use at run time. Typically, a side-by-side assembly is one to several DLLs.

Shared or private

A side-by-side assembly can be either shared or private. Shared side-by-side assemblies may be used by multiple applications that specify, in their manifests, a dependence on the assembly. Multiple versions of a side-by-side assembly can be shared by different applications that are running at the same time. A private assembly is an assembly that is deployed together with an application for the exclusive use of that application. Private assemblies are installed in the folder that contains the application's executable file or one of its subfolders.

Manifests and search order

Both isolated applications and side-by-side assemblies are described by manifests. A manifest is an XML document that can be an external file or can be embedded in an application or an assembly as a resource. The manifest file of an isolated application is used to manage the names and versions of shared side-by-side assemblies to which the application should bind at run time. The manifest of a side-by-side assembly specifies names, versions, resources, and dependent assemblies of side-by-side assemblies. For a shared side-by-side assembly, its manifest is installed in the %WINDIR%\WinSxS\Manifests\ folder. In the case of a private assembly, we recommend that you include its manifest in the DLL as a resource that has an ID equal to 1. You can also give the private assembly the same name as that of the DLL. For more information, see About Private Assemblies.

At execution time, Windows uses assembly information from the application manifest to search and load the corresponding side-by-side assembly. If an isolated application specifies an assembly dependency, the operating system first searches for the assembly among the shared assemblies in the native assembly cache in the %WINDIR%\WinSxS\

folder. If the required assembly is not found, the operating system then searches for a private assembly in a folder of the application's directory structure. For more information, see Assembly Searching Sequence.

Changing dependencies

You can change side-by-side assembly dependencies after an application has been deployed by modifying the Publisher Configuration Files and Application Configuration Files. A publisher configuration file, also known as a publisher policy file, is an XML file that globally redirects applications and assemblies from using one version of a side-by-side assembly to using another version of the same assembly. For example, you could change a dependency when a bug fix or security fix is deployed for a side-by-side assembly and you want to redirect all applications to use the fixed version. An application configuration file is an XML file that redirects a specific application from using one version of a side-by-side assembly to using another version of the same assembly. You can use an application configuration file to redirect a particular application to use a version of a side-by-side assembly that's different from the one that's defined in the publisher configuration file. For more information, see Configuration.

Visual C++ libraries

In Visual Studio 2005 and Visual Studio 2008, redistributable libraries such as ATL, MFC, CRT, Standard C++, OpenMP, and MSDIA were deployed as shared side-by-side assemblies to the native assembly cache. In the current version, the redistributable libraries use central deployment. By default, all applications that are built by using Visual Studio are built with the manifest embedded in the final binary, and the manifest describes the dependencies of the binary on the Visual C++ libraries. To understand manifest generation for C++ applications, see Understanding Manifest Generation for C/C++ Programs. A manifest is not required for applications that are statically linked to the libraries that they use, or that use local deployment. For more information about deployment, see Deployment in Visual C++.

See also

Building C/C++ Isolated Applications and Side-by-side Assemblies

Building C/C++ Isolated Applications

Article • 08/03/2021

An isolated application depends only on side-by-side assemblies and binds to its dependencies using a manifest. It is not required for your application to be fully isolated in order to run properly on Windows; however, by investing in making your application fully isolated, you may save time if you need to service your application in the future. For more information on the advantages of making your application fully isolated, see Isolated Applications.

When you build your native C/C++ application using Visual Studio, by default the Visual Studio project system generates a manifest file that describes your application's dependencies on Visual Studio libraries. If these are the only dependencies your application has, then it becomes an isolated application as soon as it is rebuilt with Visual Studio. If your application is using other libraries at runtime, then you may need to rebuild those libraries as side-by-side assemblies following the steps described in Building C/C++ Side-by-side Assemblies.

See also

Concepts of Isolated Applications and Side-by-side Assemblies
Building C/C++ Isolated Applications and Side-by-side Assemblies

Building C/C++ Side-by-side Assemblies

Article • 08/03/2021

A side-by-side assembly is a collection of resources—a group of DLLs, windows classes, COM servers, type libraries, or interfaces—available for an application to use at runtime. The primary advantage of repackaging DLLs in assemblies is that multiple versions of assemblies can be used by applications at the same time and it is possible to service currently installed assemblies in case of an update release.

A C++ application may use one or several DLLs in different parts of the application. At runtime, the DLLs are loaded into the main process and the required code is executed. The application relies on the operating system to locate the requested DLLs, understand what other dependent DLLs have to be loaded and then load them together with the requested DLL. On Windows operating systems versions earlier than Windows XP, Windows Server 2003, and Windows Vista, the operating system loader searches for dependent DLLs in either the application's local folder or another folder specified on the system path. On Windows XP, Windows Server 2003, and Windows Vista, the operating system loader can also search for dependent DLLs using a manifest file and search for side-by-side assemblies that contain these DLLs.

By default, when a DLL is built with Visual Studio, it has an application manifest embedded as an RT_MANIFEST resource with ID equal to 2. Just as for an executable, this manifest describes dependencies of this DLL on other assemblies. This assumes that the DLL is not part of a side-by-side assembly and applications that depend on this DLL are not going to use an application manifest to load it, but instead rely on the operating system loader to find this DLL on the system path.

> **Note**
>
> It is important for a DLL that uses an application manifest to have the manifest embedded as a resource with ID equal to 2. If the DLL is dynamically loaded at runtime (for example, using the **LoadLibrary** function), the operating system loader loads dependent assemblies specified in the DLL's manifest. An external application manifest for DLLs is not checked during a `LoadLibrary` call. If the manifest is not embedded, the loader may attempt to load incorrect versions of assemblies or fail to find to find dependent assemblies.

One or several related DLLs can be repackaged into a side-by-side assembly with a corresponding assembly manifest, which describes which files form the assembly as well as the dependence of the assembly on other side-by-side assemblies.

> **Note**
>
> If an assembly contains one DLL, it is recommended to embed the assembly manifest into this DLL as a resource with ID equal to 1, and give the private assembly the same name as the DLL. For example, if the name of the DLL is mylibrary.dll, the value of the name attribute used in the <assemblyIdentity> element of the manifest may also be mylibrary. In some cases, when a library has an extension other than .dll (for example, an MFC ActiveX Controls project creates an .ocx library) an external assembly manifest can be created. In this case, the name of the assembly and its manifest must be different than the name of the DLL (for example, MyAssembly, MyAssembly.manifest, and mylibrary.ocx). However it is still recommended to rename such libraries to have the extension.dll and embed the manifest as a resource to reduce the future maintenance cost of this assembly. For more information about how the operating system searches for private assemblies, see **Assembly Searching Sequence**.

This change may allow deployment of corresponding DLLs as a private assembly in an application local folder or as a shared assembly in the WinSxS assembly cache. Several steps have to be followed in order to achieve correct runtime behavior of this new assembly; they are described in Guidelines for Creating Side-by-side Assemblies. After an assembly is correctly authored it can deployed as either a shared or private assembly together with an application that depends on it. When installing side-by-side assemblies as a shared assembly, you may either follow the guidelines outlined in Installing Win32 Assemblies for Side-by-Side Sharing on Windows XP or use merge modules. When installing side-by-side assemblies as a private assembly, you may just copy the corresponding DLL, resources and assembly manifest as part of the installation process to the application local folder on the target computer, ensuring that this assembly can be found by the loader at runtime (see Assembly Searching Sequence). Another way is to use Windows Installer and follow the guidelines outlined in Installing Win32 Assemblies for the Private Use of an Application on Windows XP.

See also

Building C/C++ Isolated Applications
Building C/C++ Isolated Applications and Side-by-side Assemblies

How to: Build Registration-Free COM Components

Article • 08/03/2021

Registration-free COM components are COM components that have manifests built into the DLLs.

To build manifests into COM components

1. Open the project property pages for the COM component.
2. Expand the **Configuration Properties** node, and then expand the **Manifest Tool** node.
3. Select the **Input and Output** property page, and then set the **Embed Manifest** property equal to **Yes**.
4. Click **OK**.
5. Build the solution.

See also

How to: Build Isolated Applications to Consume COM Components

How to: Build Isolated Applications to Consume COM Components

Article • 08/03/2021

Isolated applications are applications that have manifests built into the program. You can create isolated applications to consume COM components.

To add COM references to manifests of isolated applications

1. Open the project property pages for the isolated application.
2. Expand the **Configuration Properties** node, and then expand the **Manifest Tool** node.
3. Select the **Isolated COM** property page, and then set the **Component File Name** property to the name of the COM component that you want the isolated application to consume.
4. Click **OK**.

To build manifests into isolated applications

1. Open the project property pages for the isolated application.
2. Expand the **Configuration Properties** node, and then expand the **Manifest Tool** node.
3. Select the **Input and Output** property page, and then set the **Embed Manifest** property equal to **Yes**.
4. Click **OK**.
5. Build the solution.

See also

Isolated Applications
About Side-by-Side Assemblies

Understanding manifest generation for C/C++ programs

Article • 06/14/2022

A manifest is an XML document that uniquely identifies an assembly. It contains information used for binding and activation, such as COM classes, interfaces, and type libraries. A manifest can be an external XML file or a resource embedded inside an application or an assembly. The manifest of an isolated application is used to manage the names and versions of shared side-by-side assemblies the application should bind to at run time. The manifest of a side-by-side assembly specifies its dependencies on names, versions, resources, and other assemblies.

There are two ways to create a manifest for an isolated application or a side-by-side assembly. First, the author of the assembly can manually create a manifest file by following the rules and naming requirements. For more information, see Manifest files reference. Alternatively, if a program only depends on MSVC assemblies such as CRT, MFC, ATL or others, then the linker can generate a manifest automatically.

The headers of MSVC libraries contain assembly information, and when the libraries are included in application code, this assembly information is used by the linker to form a manifest for the final binary. By default, the linker doesn't embed the manifest file inside the binary. Having a manifest as an external file may not work for all scenarios. For example, it's recommended that private assemblies have embedded manifests. In command line builds such as ones that use NMAKE to build code, you can use the /MANIFEST:EMBED linker option to embed the manifest. Alternatively, a manifest can be embedded using the manifest tool. For more information, see Manifest generation at the command line. When you build in Visual Studio, a manifest can be embedded by setting a property for the manifest tool in the **Project Properties** dialog, as described in the next section.

Manifest generation in Visual Studio

You can tell Visual Studio to generate a manifest file for a particular project in the project's **Property Pages** dialog. Under **Configuration Properties**, select **Linker** > **Manifest File** > **Generate Manifest**. By default, the project properties of new projects are set to generate a manifest file. However it's possible to disable generation of the manifest for a project by using the **Generate Manifest** property of the project. When this property is set to **Yes**, the manifest for the project is generated. Otherwise the linker

ignores assembly information when resolving dependencies of the application code, and doesn't generate the manifest.

The build system in Visual Studio allows the manifest to be embedded in the final binary application file, or generated as an external file. This behavior is controlled by the **Embed Manifest** option in the **Project Properties** dialog. To set this property, open the **Manifest Tool** node, then select **Input and Output**. If the manifest isn't embedded, it's generated as an external file and saved in the same directory as the final binary. If the manifest is embedded, Visual Studio embeds the final manifests using the following process:

1. After the source code is compiled to object files, the linker collects dependent assembly information. While it links the final binary, the linker generates an intermediate manifest that's used later to generate the final manifest.

2. After the intermediate manifest and linking are finished, the manifest tool merges a final manifest and saves it as an external file.

3. The project build system then detects whether the manifest generated by the manifest tool contains different information than the manifest already embedded in the binary.

4. If the manifest embedded in the binary is different from the manifest generated by the manifest tool, or the binary doesn't contain an embedded manifest, Visual Studio invokes the linker one more time to embed the external manifest file inside the binary as a resource.

5. If the manifest embedded in the binary is the same as the manifest generated by the manifest tool, the build continues to the next build steps.

The manifest is embedded inside the final binary as a text resource. You can view it by opening the final binary as a file in Visual Studio. To ensure that the manifest points to the correct libraries, follow the steps described in Understanding the dependencies of a Visual C++ application. Or, follow the suggestions described in the Troubleshooting article.

Manifest generation at the command line

When you build C/C++ applications from the command line using NMAKE or similar tools, the manifest is generated after the linker has processed all object files and built the final binary. The linker collects assembly information stored in the object files and combines this information into a final manifest file. By default, the linker generates a file named `<binary_name>.<extension>.manifest` to describe the final binary. The linker can

embed a manifest file inside the binary by specifying the /MANIFEST:EMBED linker option.

There are several other ways to embed a manifest inside the final binary, such as using the Manifest tool (mt.exe) or compiling the manifest into a resource file. You must follow specific rules when you embed a manifest to enable features such as incremental linking, signing, and Edit and Continue. These rules and other options are discussed in the next section.

How to embed a manifest inside a C/C++ application

We recommended that you embed the manifest of your application or library inside the final binary. This approach guarantees correct runtime behavior in most scenarios. By default, Visual Studio tries to embed the manifest when it builds a project. However, if you build your application by using NMAKE, you have to make some changes to the makefile. This section shows how to change the makefiles so that it automatically embeds the manifest inside the final binary.

Two approaches

There are two ways to embed the manifest inside an application or library.

1. If you aren't doing an incremental build, you can directly embed the manifest using a command line similar to the following as a post-build step:

 Windows Command Prompt

   ```
   mt.exe -manifest MyApp.exe.manifest -outputresource:MyApp.exe;1
   ```

 or

 Windows Command Prompt

   ```
   mt.exe -manifest MyLibrary.dll.manifest -outputresource:MyLibrary.dll;2
   ```

 Use 1 for an EXE and 2 for a DLL.

2. If you're doing an incremental build, use the following steps:

 - Link the binary to generate the `MyApp.exe.manifest` file.

- Convert the manifest to a resource file.
- Relink (incrementally) to embed the manifest resource into the binary.

The following examples show how to change makefiles to incorporate both techniques.

Makefiles (Before)

Consider the NMAKE script for *MyApp.exe*, a simple application built from one file:

makefile

```
# build MyApp.exe
!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /INCREMENTAL
!else
CPPFLAGS=$(CPPFLAGS) /MD
!endif

MyApp.exe : MyApp.obj
    link $** /out:$@ $(LFLAGS)

MyApp.obj : MyApp.cpp

clean :
    del MyApp.obj MyApp.exe
```

If this script is run unchanged with Visual Studio, it successfully creates *MyApp.exe*. It also creates the external manifest file *MyApp.exe.manifest*, for use by the operating system to load dependent assemblies at runtime.

The NMAKE script for *MyLibrary.dll* looks similar:

makefile

```
# build MyLibrary.dll
!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /DLL /INCREMENTAL

!else
CPPFLAGS=$(CPPFLAGS) /MD
LFLAGS=$(LFLAGS) /DLL

!endif

MyLibrary.dll : MyLibrary.obj
    link $** /out:$@ $(LFLAGS)
```

```
MyLibrary.obj : MyLibrary.cpp

clean :
    del MyLibrary.obj MyLibrary.dll
```

Makefiles (After)

To build with embedded manifests, you have to make four small changes to the original makefiles. For the *MyApp.exe* makefile:

makefile

```
# build MyApp.exe
!include makefile.inc
#^^^^^^^^^^^^^^^^^^^^ Change #1. (Add full path if necessary.)

!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /INCREMENTAL
!else
CPPFLAGS=$(CPPFLAGS) /MD
!endif

MyApp.exe : MyApp.obj
    link $** /out:$@ $(LFLAGS)
    $(_VC_MANIFEST_EMBED_EXE)
#^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Change #2

MyApp.obj : MyApp.cpp

clean :
    del MyApp.obj MyApp.exe
    $(_VC_MANIFEST_CLEAN)
#^^^^^^^^^^^^^^^^^^^^^^^^ Change #3

!include makefile.target.inc
#^^^^^^^^^^^^^^^^^^^^^^^^^^ Change #4. (Add full path if necessary.)
```

For the MyLibrary.dll makefile:

makefile

```
# build MyLibrary.dll
!include makefile.inc
#^^^^^^^^^^^^^^^^^^^^ Change #1. (Add full path if necessary.)

!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /DLL /INCREMENTAL
```

```
!else
CPPFLAGS=$(CPPFLAGS) /MD
LFLAGS=$(LFLAGS) /DLL

!endif

MyLibrary.dll : MyLibrary.obj
    link $** /out:$@ $(LFLAGS)
    $(_VC_MANIFEST_EMBED_DLL)
#^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Change #2.

MyLibrary.obj : MyLibrary.cpp

clean :
    del MyLibrary.obj MyLibrary.dll
    $(_VC_MANIFEST_CLEAN)
#^^^^^^^^^^^^^^^^^^^^^^^^ Change #3.

!include makefile.target.inc
#^^^^^^^^^^^^^^^^^^^^^^^^^^ Change #4. (Add full path if necessary.)
```

The makefiles now include two files that do the real work, `makefile.inc` and `makefile.target.inc`.

Create `makefile.inc` and copy the following content into it:

makefile

```
# makefile.inc -- Include this file into existing makefile at the very top.

# _VC_MANIFEST_INC specifies whether build is incremental (1 - incremental).
# _VC_MANIFEST_BASENAME specifies name of a temporary resource file.

!if "$(DEBUG)" == "1"
CPPFLAGS=$(CPPFLAGS) /MDd
LFLAGS=$(LFLAGS) /INCREMENTAL
_VC_MANIFEST_INC=1
_VC_MANIFEST_BASENAME=__VC90.Debug

!else
CPPFLAGS=$(CPPFLAGS) /MD
_VC_MANIFEST_INC=0
_VC_MANIFEST_BASENAME=__VC90

!endif

########################################################
# Specifying name of temporary resource file used only in incremental
builds:

!if "$(_VC_MANIFEST_INC)" == "1"
```

```
_VC_MANIFEST_AUTO_RES=$(_VC_MANIFEST_BASENAME).auto.res
!else
_VC_MANIFEST_AUTO_RES=
!endif

#########################################################
# _VC_MANIFEST_EMBED_EXE - command to embed manifest in EXE:

!if "$(_VC_MANIFEST_INC)" == "1"

#MT_SPECIAL_RETURN=1090650113
#MT_SPECIAL_SWITCH=-notify_resource_update
MT_SPECIAL_RETURN=0
MT_SPECIAL_SWITCH=
_VC_MANIFEST_EMBED_EXE= \
if exist $@.manifest mt.exe -manifest $@.manifest -
out:$(_VC_MANIFEST_BASENAME).auto.manifest $(MT_SPECIAL_SWITCH) & \
if "%ERRORLEVEL%" == "$(MT_SPECIAL_RETURN)" \
rc /r $(_VC_MANIFEST_BASENAME).auto.rc & \
link $** /out:$@ $(LFLAGS)

!else

_VC_MANIFEST_EMBED_EXE= \
if exist $@.manifest mt.exe -manifest $@.manifest -outputresource:$@;1

!endif

#########################################################
# _VC_MANIFEST_CLEAN - command to clean resources files generated
temporarily:

!if "$(_VC_MANIFEST_INC)" == "1"

_VC_MANIFEST_CLEAN=-del $(_VC_MANIFEST_BASENAME).auto.res \
    $(_VC_MANIFEST_BASENAME).auto.rc \
    $(_VC_MANIFEST_BASENAME).auto.manifest

!else

_VC_MANIFEST_CLEAN=

!endif

# End of makefile.inc
#########################################################
```

Now create *makefile.target.inc* and copy the following content into it:

makefile

```
# makefile.target.inc - include this at the very bottom of the existing
makefile
```

```
#########################################################
# Commands to generate initial empty manifest file and the RC file
# that references it, and for generating the .res file:

$(_VC_MANIFEST_BASENAME).auto.res : $(_VC_MANIFEST_BASENAME).auto.rc

$(_VC_MANIFEST_BASENAME).auto.rc : $(_VC_MANIFEST_BASENAME).auto.manifest
    type <<$@
#include <winuser.h>
1RT_MANIFEST"$(_VC_MANIFEST_BASENAME).auto.manifest"
<< KEEP

$(_VC_MANIFEST_BASENAME).auto.manifest :
    type <<$@
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>
<assembly xmlns='urn:schemas-microsoft-com:asm.v1' manifestVersion='1.0'>
</assembly>
<< KEEP

# end of makefile.target.inc
```

See also

Building C/C++ isolated applications and side-by-side assemblies
Concepts of isolated applications and side-by-side assemblies
Troubleshooting C/C++ isolated applications and side-by-side assemblies
/INCREMENTAL (Link incrementally)
/MANIFEST (Create side-by-side assembly manifest)
Strong Name assemblies (Assembly signing) (C++/CLI)
Edit and Continue

Troubleshooting C/C++ Isolated Applications and Side-by-side Assemblies

Article • 08/03/2021

Loading a C/C++ application can fail if dependent libraries cannot be found. This article describes some common reasons why a C/C++ application fails to load, and suggests steps to resolve the problems.

If an application fails to load because it has a manifest that specifies a dependency on a side-by-side assembly, and the assembly is not installed as a private assembly in the same folder as the executable nor in the native assembly cache in the %WINDIR%\WinSxS\ folder, one of the following error messages might be displayed, depending on the version of Windows on which you try to run the app.

- The application failed to initialize properly (0xc0000135).
- This application has failed to start because the application configuration is incorrect. Reinstalling the application may fix this problem.
- The system cannot execute the specified program.

If your application has no manifest and depends on a DLL that Windows can't find in the typical search locations, an error message that resembles this one might be displayed:

- This application has failed to start because *a required DLL* was not found. Re-installing the application may fix this problem.

If your application is deployed on a computer that doesn't have Visual Studio, and it crashes with error messages that resemble the previous ones, check these things:

1. Follow the steps that are described in Understanding the Dependencies of a Visual C++ Application. The dependency walker can show most dependencies for an application or DLL. If you observe that some DLLs are missing, install them on the computer on which you are trying to run your application.
2. The operating system loader uses the application manifest to load assemblies that the application depends on. The manifest can either be embedded in the binary as a resource, or installed as a separate file in the application folder. To check whether the manifest is embedded in the binary, open the binary in Visual Studio and look for RT_MANIFEST in its list of resources. If you can't find an embedded manifest,

look in the application folder for a file that's named something like <binary_name>.<extension>.manifest.

3. If your application depends on side-by-side assemblies and a manifest is not present, you have to ensure that the linker generates a manifest for your project. Check the linker option **Generate manifest** in the **Project Properties** dialog box for the project.

4. If the manifest is embedded in the binary, ensure that the ID of RT_MANIFEST is correct for this type of the binary. For more information about which resource ID to use, see Using Side-by-Side Assemblies as a Resource (Windows). If the manifest is in a separate file, open it in an XML editor or text editor. For more information about manifests and rules for deployment, see Manifests.

 > **Note**
 >
 > If both an embedded manifest and a separate manifest file are present, the operating system loader uses the embedded manifest and ignores the separate file. However, on Windows XP, the opposite is true—the separate manifest file is used and the embedded manifest is ignored.

5. We recommend that you embed a manifest in every DLL because external manifests are ignored when a DLL is loaded though a `LoadLibrary` call. For more information, see Assembly manifests.

6. Check that all assemblies that are enumerated in the manifest are correctly installed on the computer. Each assembly is specified in the manifest by its name, version number, and processor architecture. If your application depends on side-by-side assemblies, check that these assemblies are correctly installed on the computer so that the operating system loader can find them, as described in Assembly Searching Sequence. Remember that 64-bit assemblies cannot be loaded in 32-bit processes and cannot be executed on 32-bit operating systems.

Example

Assume we have an application, appl.exe, that's built by using Visual C++. The application manifest either is embedded in appl.exe as the binary resource RT_MANIFEST, which has an ID equal to 1, or is stored as the separate file appl.exe.manifest. The content of this manifest resembles this:

```
```

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Fabrikam.SxS.Library"
version="2.0.20121.0" processorArchitecture="x86"
publicKeyToken="1fc8b3b9a1e18e3e"></assemblyIdentity>
    </dependentAssembly>
  </dependency>
</assembly>
```

To the operating system loader, this manifest says that appl.exe depends on an assembly named Fabrikam.SxS.Library, version 2.0.20121.0, that's built for a 32-bit x86 processor architecture. The dependent side-by-side assembly can be installed either as a shared assembly or as a private assembly.

The assembly manifest for a shared assembly is installed in the %WINDIR%\WinSxS\Manifests\ folder. It identifies the assembly and lists its contents—that is, the DLLs that are part of the assembly:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
   <noInheritable/>
   <assemblyIdentity type="win32" name="Fabrikam.SxS.Library"
version="2.0.20121.0" processorArchitecture="x86"
publicKeyToken="1fc8b3b9a1e18e3e"/>
   <file name="Fabrikam.Main.dll"
hash="3ca5156e8212449db6c622c3d10f37d9adb1ab12" hashalg="SHA1"/>
   <file name="Fabrikam.Helper.dll"
hash="92cf8a9bb066aea821d324ca4695c69e55b2d1c2" hashalg="SHA1"/>
</assembly>
```

Side-by-side assemblies can also use publisher configuration files—also known as policy files—to globally redirect applications and assemblies to use one version of a side-by-side assembly instead of another version of the same assembly. You can check the policies for a shared assembly in the %WINDIR%\WinSxS\Policies\ folder. Here is an example policy file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">

   <assemblyIdentity type="win32-policy"
name="policy.2.0.Fabrikam.SxS.Library" version="2.0.20121.0"
processorArchitecture="x86" publicKeyToken="1fc8b3b9a1e18e3e"/>
   <dependency>
```

```
      <dependentAssembly>
         <assemblyIdentity type="win32" name="Fabrikam.SxS.Library"
processorArchitecture="x86" publicKeyToken="1fc8b3b9a1e18e3e"/>
         <bindingRedirect oldVersion="2.0.10000.0-2.0.20120.99"
newVersion="2.0.20121.0"/>
      </dependentAssembly>
   </dependency>
</assembly>
```

This policy file specifies that any application or assembly that asks for version 2.0.10000.0 of this assembly should instead use version 2.0.20121.0, which is the current version that's installed on the system. If a version of the assembly that's mentioned in the application manifest is specified in the policy file, the loader looks for a version of this assembly that's specified in the manifest in the %WINDIR%\WinSxS\ folder, and if this version is not installed, load fails. And if assembly version 2.0.20121.0 is not installed, load fails for applications that ask for assembly version 2.0.10000.0.

However, the assembly can also be installed as a private side-by-side assembly in the installed application folder. If the operating system fails to find the assembly as a shared assembly, it looks for it as a private assembly, in the following order:

1. Check the application folder for a manifest file that has the name <assemblyName>.manifest. In this example, the loader tries to find Fabrikam.SxS.Library.manifest in the folder that contains appl.exe. If it finds the manifest, the loader loads the assembly from the application folder. If the assembly is not found, load fails.

2. Try to open the \<assemblyName>\ folder in the folder that contains appl.exe, and if \<assemblyName>\ exists, try to load a manifest file that has the name <assemblyName>.manifest from this folder. If the manifest is found, the loader loads the assembly from the \<assemblyName>\ folder. If the assembly is not found, load fails.

For more information about how the loader searches for dependent assemblies, see Assembly Searching Sequence. If the loader fails to find a dependent assembly as a private assembly, load fails and the message "The system cannot execute the specified program" is displayed. To resolve this error, make sure that dependent assemblies—and DLLs that are part of them—are installed on the computer as either private or shared assemblies.

See also

Concepts of Isolated Applications and Side-by-side Assemblies
Building C/C++ Isolated Applications and Side-by-side Assemblies

Configure C++ projects for 64-bit, x64 targets

Article • 08/03/2021

This section contains topics about targeting 64-bit x64 hardware with the Visual C++ build tools.

In This Section

- How to: Configure Visual C++ Projects to Target 64-Bit, x64 Platforms
- How to: Enable a 64-bit, x64-hosted MSVC toolset on the command line
- Common Visual C++ 64-bit Migration Issues
- x64 Software Conventions

Related Sections

.NET Framework 64-bit Applications

align

/clr (Common Language Runtime Compilation)

/favor (Optimize for Architecture Specifics)

Programming Guide for 64-bit Windows

MASM for x64 (ml64.exe)

x64 (amd64) Intrinsics List

See also

Projects and build systems

How to: Configure Visual Studio C++ projects to Target 64-Bit, x64 Platforms

Article • 08/03/2021

You can use the project configurations in the Visual Studio IDE to set up C++ applications to target 64-bit, x64 platforms. You can also migrate Win32 project settings into a 64-bit project configuration.

To set up C++ applications to target 64-bit platforms

1. Open the C++ project that you want to configure.
2. Open the property pages for that project. For more information, see Set C++ compiler and build properties in Visual Studio.

 > **Note**
 >
 > For .NET projects, make sure that the **Configuration Properties** node, or one of its child nodes, is selected in the **<Projectname> Property Pages** dialog box; otherwise, the **Configuration Manager** button remains unavailable.

3. Choose the **Configuration Manager** button to open the **Configuration Manager** dialog box.
4. In the **Active Solution Platform** drop-down list, select the **<New...>** option to open the **New Solution Platform** dialog box.
5. In the **Type or select the new platform** drop-down list, select a 64-bit target platform.

 > **Note**
 >
 > In the **New Solution Platform** dialog box, you can use the **Copy settings from** option to copy existing project settings into the new 64-bit project configuration.

6. Choose the **OK** button. The platform that you selected in the preceding step appears under **Active Solution Platform** in the **Configuration Manager** dialog box.

7. Choose the **Close** button in the **Configuration Manager** dialog box, and then choose the **OK** button in the <**Projectname**> **Property Pages** dialog box.

To copy Win32 project settings into a 64-bit project configuration

- When the **New Solution Platform** dialog box is open while you set up a project to target a 64-bit platform, in the **Copy settings from** drop-down list, select **Win32**. These project settings are automatically updated on the project level:
 - The /MACHINE linker option is set to **/MACHINE:X64**.
 - **Register Output** is turned OFF. For more information, see Linker Property Pages.
 - **Target Environment** is set to **/env x64**. For more information, see MIDL Property Pages.
 - **Validate Parameters** is cleared and reset to the default value. For more information, see MIDL Property Pages.
 - If **Debug Information Format** was set to **/ZI** in the Win32 project configuration, then it is set to **/Zi** in the 64-bit project configuration. For more information, see /Z7, /Zi, /ZI (Debug Information Format).

> **Note**
>
> None of these project properties are changed if they are overridden on the file level.

See also

Configure C++ projects for 64-bit, x64 targets
Debug 64-Bit Applications

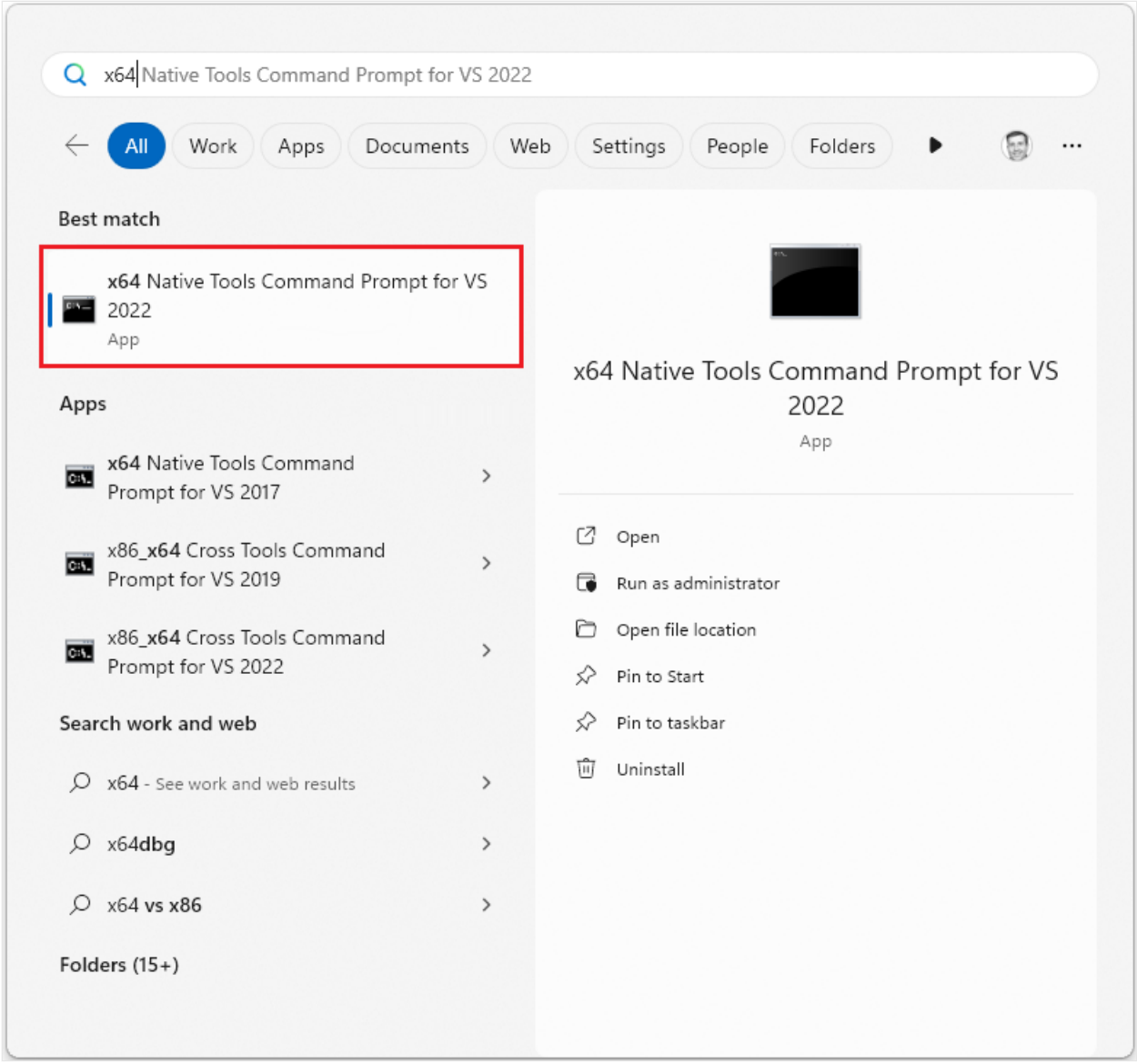
How to: Enable a 64-Bit, x64 hosted MSVC toolset on the command line

Article • 10/16/2023

Visual Studio includes C++ compilers, linkers, and other tools that you can use to create platform-specific versions of your apps that can run on 32-bit, 64-bit, or ARM-based Windows operating systems. Other optional Visual Studio workloads let you use C++ tools to target other platforms, such as iOS, Android, and Linux. The default build architecture uses 32-bit, x86-hosted tools to build 32-bit, x86-native Windows code. However, you probably have a 64-bit computer. When Visual Studio is installed on a 64-bit Windows operating system, additional developer command prompt shortcuts for the 64-bit, x64-hosted native and cross compilers are available. You can take advantage of the processor and memory space available to 64-bit code by using the 64-bit, x64-hosted toolset when you build code for x86, x64, or ARM processors.

Use a 64-bit hosted developer command prompt shortcut

To access these command prompts on Windows, on the **Start** menu type `x64` and then choose one of the x64 native or cross-tool developer command prompts.



On earlier versions of Windows, choose **Start**, expand **All Programs**, and then expand the folder for your version of **Visual Studio** (and on older versions of Visual Studio, **Visual Studio Tools**). For more information, see Developer command prompt shortcuts.

Use Vcvarsall.bat to set a 64-bit hosted build architecture

Any of the native or cross compiler tools build configurations can be used on the command line by running the vcvarsall.bat command file. This command file configures the path and environment variables that enable a particular build architecture in an existing command prompt window. For specific instructions, see Developer command file locations.

Remarks

> **Note**
>
> For information about the specific tools that are included with each Visual Studio edition, see **Visual C++ Tools and Features in Visual Studio Editions**.
>
> For information about how to use the Visual Studio IDE to create 64-bit applications, see **How to: Configure Visual C++ Projects to Target 64-Bit, x64 Platforms**.

When you install a C++ workload in the Visual Studio installer, it always installs 32-bit, x86-hosted, native and cross compiler tools to build x86 and x64 code. If you include the Universal Windows Platform workload, it also installs x86-hosted cross compiler tools to build ARM code. If you install these workloads on a 64-bit, x64 processor, you also get 64-bit native and cross compiler tools to build x86, x64, and ARM code. The 32-bit and 64-bit tools generate identical code, but the 64-bit tools support more memory for precompiled header symbols and the Whole Program Optimization (/GL and /LTCG) options. If you run into memory limits when you use the 32-bit tools, try the 64-bit tools.

See also

Configure C++ projects for 64-bit, x64 targets

Common Visual C++ 64-bit Migration Issues

Article • 08/03/2021

When you use the Microsoft C++ compiler (MSVC) to create applications to run on a 64-bit Windows operating system, you should be aware of the following issues:

- An `int` and a `long` are 32-bit values on 64-bit Windows operating systems. For programs that you plan to compile for 64-bit platforms, you should be careful not to assign pointers to 32-bit variables. Pointers are 64-bit on 64-bit platforms, and you will truncate the pointer value if you assign it to a 32-bit variable.
- `size_t`, `time_t`, and `ptrdiff_t` are 64-bit values on 64-bit Windows operating systems.
- `time_t` is a 32-bit value on 32-bit Windows operating systems in Visual Studio 2005 and earlier. `time_t` is now a 64-bit integer by default. For more information, see Time Management.

 You should be aware of where your code takes an `int` value and processes it as a `size_t` or `time_t` value. It is possible that the number could grow to be larger than a 32-bit number and data will be truncated when it is passed back to the `int` storage.

The %x (hex `int` format) `printf` modifier will not work as expected on a 64-bit Windows operating system. It will only operate on the first 32 bits of the value that is passed to it.

- Use %I32x to display a 32-bit integral type in hex format.
- Use %I64x to display a 64-bit integral type in hex format.
- The %p (hex format for a pointer) will work as expected on a 64-bit Windows operating system.

For more information, see:

- MSVC Compiler Options
- Migration Tips

See also

Configure C++ projects for 64-bit, x64 targets

Visual C++ Porting and Upgrading Guide

Overview of x64 ABI conventions

Article • 04/22/2022

This topic describes the basic application binary interface (ABI) for x64, the 64-bit extension to the x86 architecture. It covers topics such as the calling convention, type layout, stack and register usage, and more.

x64 calling conventions

Two important differences between x86 and x64 are:

- 64-bit addressing capability
- Sixteen 64-bit registers for general use.

Given the expanded register set, x64 uses the __fastcall calling convention and a RISC-based exception-handling model.

The `__fastcall` convention uses registers for the first four arguments, and the stack frame to pass more arguments. For details on the x64 calling convention, including register usage, stack parameters, return values, and stack unwinding, see x64 calling convention.

Enable x64 compiler optimization

The following compiler option helps you optimize your application for x64:

- /favor (Optimize for Architecture Specifics)

x64 type and storage layout

This section describes the storage of data types for the x64 architecture.

Scalar types

Although it's possible to access data with any alignment, align data on its natural boundary, or a multiple of its natural boundary, to avoid performance loss. Enums are constant integers and are treated as 32-bit integers. The following table describes the type definition and recommended storage for data as it pertains to alignment using the following alignment values:

- Byte - 8 bits
- Word - 16 bits
- Doubleword - 32 bits
- Quadword - 64 bits
- Octaword - 128 bits

Scalar type	C data type	Storage size (in bytes)	Recommended alignment
`INT8`	`char`	1	Byte
`UINT8`	`unsigned char`	1	Byte
`INT16`	`short`	2	Word
`UINT16`	`unsigned short`	2	Word
`INT32`	`int`, `long`	4	Doubleword
`UINT32`	`unsigned int`, `unsigned long`	4	Doubleword
`INT64`	`__int64`	8	Quadword
`UINT64`	`unsigned __int64`	8	Quadword
`FP32` (single precision)	`float`	4	Doubleword
`FP64` (double precision)	`double`	8	Quadword
`POINTER`	*	8	Quadword
`__m64`	`struct __m64`	8	Quadword
`__m128`	`struct __m128`	16	Octaword

x64 aggregate and union layout

Other types, such as arrays, structs, and unions, have stricter alignment requirements that ensure consistent aggregate and union storage and data retrieval. Here are the definitions for array, structure, and union:

- Array

 Contains an ordered group of adjacent data objects. Each object is called an *element*. All elements within an array have the same size and data type.

- Structure

 Contains an ordered group of data objects. Unlike the elements of an array, the members of a structure can have different data types and sizes.

- Union

 An object that holds any one of a set of named members. The members of the named set can be of any type. The storage allocated for a union is equal to the storage required for the largest member of that union, plus any padding required for alignment.

The following table shows the strongly recommended alignment for the scalar members of unions and structures.

Scalar Type	C Data Type	Required Alignment
`INT8`	`char`	Byte
`UINT8`	`unsigned char`	Byte
`INT16`	`short`	Word
`UINT16`	`unsigned short`	Word
`INT32`	`int`, `long`	Doubleword
`UINT32`	`unsigned int`, `unsigned long`	Doubleword
`INT64`	`__int64`	Quadword
`UINT64`	`unsigned __int64`	Quadword
`FP32` (single precision)	`float`	Doubleword
`FP64` (double precision)	`double`	Quadword
`POINTER`	*	Quadword
`__m64`	`struct __m64`	Quadword
`__m128`	`struct __m128`	Octaword

The following aggregate alignment rules apply:

- The alignment of an array is the same as the alignment of one of the elements of the array.

- The alignment of the beginning of a structure or a union is the maximum alignment of any individual member. Each member within the structure or union must be placed at its proper alignment as defined in the previous table, which may require implicit internal padding, depending on the previous member.
- Structure size must be an integral multiple of its alignment, which may require padding after the last member. Since structures and unions can be grouped in arrays, each array element of a structure or union must begin and end at the proper alignment previously determined.
- It's possible to align data in such a way as to be greater than the alignment requirements as long as the previous rules are maintained.
- An individual compiler may adjust the packing of a structure for size reasons. For example, /Zp (Struct Member Alignment) allows for adjusting the packing of structures.

x64 structure alignment examples

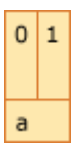
The following four examples each declare an aligned structure or union, and the corresponding figures illustrate the layout of that structure or union in memory. Each column in a figure represents a byte of memory, and the number in the column indicates the displacement of that byte. The name in the second row of each figure corresponds to the name of a variable in the declaration. The shaded columns indicate padding that is required to achieve the specified alignment.

Example 1

C

```
// Total size = 2 bytes, alignment = 2 bytes (word).

_declspec(align(2)) struct {
    short a;      // +0; size = 2 bytes
}
```

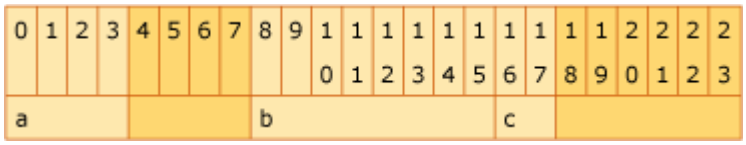


Example 2

C

```
// Total size = 24 bytes, alignment = 8 bytes (quadword).

_declspec(align(8)) struct {
    int a;       // +0; size = 4 bytes
    double b;    // +8; size = 8 bytes
    short c;     // +16; size = 2 bytes
}
```

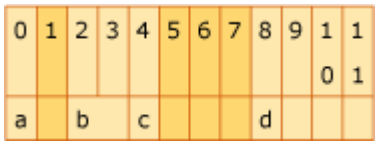


Example 3

C

```
// Total size = 12 bytes, alignment = 4 bytes (doubleword).

_declspec(align(4)) struct {
    char a;       // +0; size = 1 byte
    short b;      // +2; size = 2 bytes
    char c;       // +4; size = 1 byte
    int d;        // +8; size = 4 bytes
}
```

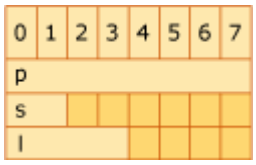


Example 4

C

```
// Total size = 8 bytes, alignment = 8 bytes (quadword).

_declspec(align(8)) union {
    char *p;      // +0; size = 8 bytes
    short s;      // +0; size = 2 bytes
    long l;       // +0; size = 4 bytes
}
```



Bitfields

Structure bit fields are limited to 64 bits and can be of type signed int, unsigned int, int64, or unsigned int64. Bit fields that cross the type boundary will skip bits to align the bitfield to the next type alignment. For example, integer bitfields may not cross a 32-bit boundary.

Conflicts with the x86 compiler

Data types that are larger than 4 bytes aren't automatically aligned on the stack when you use the x86 compiler to compile an application. Because the architecture for the x86 compiler is a 4 byte aligned stack, anything larger than 4 bytes, for example, a 64-bit integer, can't be automatically aligned to an 8-byte address.

Working with unaligned data has two implications.

- It may take longer to access unaligned locations than it takes to access aligned locations.
- Unaligned locations can't be used in interlocked operations.

If you require more strict alignment, use `__declspec(align(N))` on your variable declarations. This causes the compiler to dynamically align the stack to meet your specifications. However, dynamically adjusting the stack at run time may cause slower execution of your application.

x64 register usage

The x64 architecture provides for 16 general-purpose registers (hereafter referred to as integer registers) as well as 16 XMM/YMM registers available for floating-point use. Volatile registers are scratch registers presumed by the caller to be destroyed across a call. Nonvolatile registers are required to retain their values across a function call and must be saved by the callee if used.

Register volatility and preservation

The following table describes how each register is used across function calls:

Register	Status	Use
RAX	Volatile	Return value register
RCX	Volatile	First integer argument
RDX	Volatile	Second integer argument
R8	Volatile	Third integer argument
R9	Volatile	Fourth integer argument
R10:R11	Volatile	Must be preserved as needed by caller; used in syscall/sysret instructions
R12:R15	Nonvolatile	Must be preserved by callee
RDI	Nonvolatile	Must be preserved by callee
RSI	Nonvolatile	Must be preserved by callee
RBX	Nonvolatile	Must be preserved by callee
RBP	Nonvolatile	May be used as a frame pointer; must be preserved by callee
RSP	Nonvolatile	Stack pointer
XMM0, YMM0	Volatile	First FP argument; first vector-type argument when `__vectorcall` is used
XMM1, YMM1	Volatile	Second FP argument; second vector-type argument when `__vectorcall` is used
XMM2, YMM2	Volatile	Third FP argument; third vector-type argument when `__vectorcall` is used
XMM3, YMM3	Volatile	Fourth FP argument; fourth vector-type argument when `__vectorcall` is used
XMM4, YMM4	Volatile	Must be preserved as needed by caller; fifth vector-type argument when `__vectorcall` is used
XMM5, YMM5	Volatile	Must be preserved as needed by caller; sixth vector-type argument when `__vectorcall` is used
XMM6:XMM15, YMM6:YMM15	Nonvolatile (XMM), Volatile (upper half of YMM)	Must be preserved by callee. YMM registers must be preserved as needed by caller.

On function exit and on function entry to C Runtime Library calls and Windows system calls, the direction flag in the CPU flags register is expected to be cleared.

Stack usage

For details on stack allocation, alignment, function types and stack frames on x64, see x64 stack usage.

Prolog and epilog

Every function that allocates stack space, calls other functions, saves nonvolatile registers, or uses exception handling must have a prolog whose address limits are described in the unwind data associated with the respective function table entry, and epilogs at each exit to a function. For details on the required prolog and epilog code on x64, see x64 prolog and epilog.

x64 exception handling

For information on the conventions and data structures used to implement structured exception handling and C++ exception handling behavior on the x64, see x64 exception handling.

Intrinsics and inline assembly

One of the constraints for the x64 compiler is no inline assembler support. This means that functions that can't be written in C or C++ will either have to be written as subroutines or as intrinsic functions supported by the compiler. Certain functions are performance sensitive while others aren't. Performance-sensitive functions should be implemented as intrinsic functions.

The intrinsics supported by the compiler are described in Compiler intrinsics.

x64 image format

The x64 executable image format is PE32+. Executable images (both DLLs and EXEs) are restricted to a maximum size of 2 gigabytes, so relative addressing with a 32-bit displacement can be used to address static image data. This data includes the import address table, string constants, static global data, and so on.

See also

Calling Conventions

x64 calling convention

Article • 05/18/2022

This section describes the standard processes and conventions that one function (the caller) uses to make calls into another function (the callee) in x64 code.

For more information on the `__vectorcall` calling convention, see __vectorcall.

Calling convention defaults

The x64 Application Binary Interface (ABI) uses a four-register fast-call calling convention by default. Space is allocated on the call stack as a shadow store for callees to save those registers.

There's a strict one-to-one correspondence between a function call's arguments and the registers used for those arguments. Any argument that doesn't fit in 8 bytes, or isn't 1, 2, 4, or 8 bytes, must be passed by reference. A single argument is never spread across multiple registers.

The x87 register stack is unused. It may be used by the callee, but consider it volatile across function calls. All floating point operations are done using the 16 XMM registers.

Integer arguments are passed in registers RCX, RDX, R8, and R9. Floating point arguments are passed in XMM0L, XMM1L, XMM2L, and XMM3L. 16-byte arguments are passed by reference. Parameter passing is described in detail in Parameter passing. These registers, and RAX, R10, R11, XMM4, and XMM5, are considered *volatile*, or potentially changed by a callee on return. Register usage is documented in detail in x64 register usage and Caller/callee saved registers.

For prototyped functions, all arguments are converted to the expected callee types before passing. The caller is responsible for allocating space for the callee's parameters. The caller must always allocate sufficient space to store four register parameters, even if the callee doesn't take that many parameters. This convention simplifies support for unprototyped C-language functions and vararg C/C++ functions. For vararg or unprototyped functions, any floating point values must be duplicated in the corresponding general-purpose register. Any parameters beyond the first four must be stored on the stack after the shadow store before the call. Vararg function details can be found in Varargs. Unprototyped function information is detailed in Unprototyped functions.

Alignment

Most structures are aligned to their natural alignment. The primary exceptions are the stack pointer and `malloc` or `alloca` memory, which are 16-byte aligned to aid performance. Alignment above 16 bytes must be done manually. Since 16 bytes is a common alignment size for XMM operations, this value should work for most code. For more information about structure layout and alignment, see x64 type and storage layout. For information about the stack layout, see x64 stack usage.

Unwindability

Leaf functions are functions that don't change any non-volatile registers. A non-leaf function may change non-volatile RSP, for example, by calling a function. Or, it could change RSP by allocating additional stack space for local variables. To recover non-volatile registers when an exception is handled, non-leaf functions are annotated with static data. The data describes how to properly unwind the function at an arbitrary instruction. This data is stored as *pdata*, or procedure data, which in turn refers to *xdata*, the exception handling data. The xdata contains the unwinding information, and can point to additional pdata or an exception handler function.

Prologs and epilogs are highly restricted so that they can be properly described in xdata. The stack pointer must remain 16-byte aligned in any region of code that isn't part of an epilog or prolog, except within leaf functions. Leaf functions can be unwound simply by simulating a return, so pdata and xdata aren't required. For details about the proper structure of function prologs and epilogs, see x64 prolog and epilog. For more information about exception handling, and the exception handling and unwinding of pdata and xdata, see x64 exception handling.

Parameter passing

By default, the x64 calling convention passes the first four arguments to a function in registers. The registers used for these arguments depend on the position and type of the argument. Remaining arguments get pushed on the stack in right-to-left order.

Integer valued arguments in the leftmost four positions are passed in left-to-right order in RCX, RDX, R8, and R9, respectively. The fifth and higher arguments are passed on the stack as previously described. All integer arguments in registers are right-justified, so the callee can ignore the upper bits of the register and access only the portion of the register necessary.

Any floating-point and double-precision arguments in the first four parameters are passed in XMM0 - XMM3, depending on position. Floating-point values are only placed in the integer registers RCX, RDX, R8, and R9 when there are varargs arguments. For details, see Varargs. Similarly, the XMM0 - XMM3 registers are ignored when the corresponding argument is an integer or pointer type.

__m128 types, arrays, and strings are never passed by immediate value. Instead, a pointer is passed to memory allocated by the caller. Structs and unions of size 8, 16, 32, or 64 bits, and `__m64` types, are passed as if they were integers of the same size. Structs or unions of other sizes are passed as a pointer to memory allocated by the caller. For these aggregate types passed as a pointer, including `__m128`, the caller-allocated temporary memory must be 16-byte aligned.

Intrinsic functions that don't allocate stack space, and don't call other functions, sometimes use other volatile registers to pass additional register arguments. This optimization is made possible by the tight binding between the compiler and the intrinsic function implementation.

The callee is responsible for dumping the register parameters into their shadow space if needed.

The following table summarizes how parameters are passed, by type and position from the left:

Parameter type	fifth and higher	fourth	third	second	leftmost
floating-point	stack	XMM3	XMM2	XMM1	XMM0
integer	stack	R9	R8	RDX	RCX
Aggregates (8, 16, 32, or 64 bits) and `__m64`	stack	R9	R8	RDX	RCX
Other aggregates, as pointers	stack	R9	R8	RDX	RCX
`__m128`, as a pointer	stack	R9	R8	RDX	RCX

Example of argument passing 1 - all integers

C++

```
func1(int a, int b, int c, int d, int e, int f);
// a in RCX, b in RDX, c in R8, d in R9, f then e pushed on stack
```

Example of argument passing 2 - all floats

```
C++

func2(float a, double b, float c, double d, float e, float f);
// a in XMM0, b in XMM1, c in XMM2, d in XMM3, f then e pushed on stack
```

Example of argument passing 3 - mixed ints and floats

```
C++

func3(int a, double b, int c, float d, int e, float f);
// a in RCX, b in XMM1, c in R8, d in XMM3, f then e pushed on stack
```

Example of argument passing 4 - `__m64`, `__m128`, and aggregates

```
C++

func4(__m64 a, __m128 b, struct c, float d, __m128 e, __m128 f);
// a in RCX, ptr to b in RDX, ptr to c in R8, d in XMM3,
// ptr to f pushed on stack, then ptr to e pushed on stack
```

Varargs

If parameters are passed via varargs (for example, ellipsis arguments), then the normal register parameter passing convention applies. That convention includes spilling the fifth and later arguments to the stack. It's the callee's responsibility to dump arguments that have their address taken. For floating-point values only, both the integer register and the floating-point register must contain the value, in case the callee expects the value in the integer registers.

Unprototyped functions

For functions not fully prototyped, the caller passes integer values as integers and floating-point values as double precision. For floating-point values only, both the integer register and the floating-point register contain the float value in case the callee expects the value in the integer registers.

```
C++
```

```
func1();
func2() {   // RCX = 2, RDX = XMM1 = 1.0, and R8 = 7
   func1(2, 1.0, 7);
}
```

Return values

A scalar return value that can fit into 64 bits, including the `__m64` type, is returned through RAX. Non-scalar types including floats, doubles, and vector types such as __m128, __m128i, __m128d are returned in XMM0. The state of unused bits in the value returned in RAX or XMM0 is undefined.

User-defined types can be returned by value from global functions and static member functions. To return a user-defined type by value in RAX, it must have a length of 1, 2, 4, 8, 16, 32, or 64 bits. It must also have no user-defined constructor, destructor, or copy assignment operator. It can have no private or protected non-static data members, and no non-static data members of reference type. It can't have base classes or virtual functions. And, it can only have data members that also meet these requirements. (This definition is essentially the same as a C++03 POD type. Because the definition has changed in the C++11 standard, we don't recommend using `std::is_pod` for this test.) Otherwise, the caller must allocate memory for the return value and pass a pointer to it as the first argument. The remaining arguments are then shifted one argument to the right. The same pointer must be returned by the callee in RAX.

These examples show how parameters and return values are passed for functions with the specified declarations:

Example of return value 1 - 64-bit result

Output

```
__int64 func1(int a, float b, int c, int d, int e);
// Caller passes a in RCX, b in XMM1, c in R8, d in R9, e pushed on stack,
// callee returns __int64 result in RAX.
```

Example of return value 2 - 128-bit result

Output

```
__m128 func2(float a, double b, int c, __m64 d);
// Caller passes a in XMM0, b in XMM1, c in R8, d in R9,
```

```
// callee returns __m128 result in XMM0.
```

Example of return value 3 - user type result by pointer

Output

```
struct Struct1 {
   int j, k, l;    // Struct1 exceeds 64 bits.
};
Struct1 func3(int a, double b, int c, float d);
// Caller allocates memory for Struct1 returned and passes pointer in RCX,
// a in RDX, b in XMM2, c in R9, d pushed on the stack;
// callee returns pointer to Struct1 result in RAX.
```

Example of return value 4 - user type result by value

Output

```
struct Struct2 {
   int j, k;    // Struct2 fits in 64 bits, and meets requirements for
return by value.
};
Struct2 func4(int a, double b, int c, float d);
// Caller passes a in RCX, b in XMM1, c in R8, and d in XMM3;
// callee returns Struct2 result by value in RAX.
```

Caller/callee saved registers

The x64 ABI considers the registers RAX, RCX, RDX, R8, R9, R10, R11, and XMM0-XMM5 volatile. When present, the upper portions of YMM0-YMM15 and ZMM0-ZMM15 are also volatile. On AVX512VL, the ZMM, YMM, and XMM registers 16-31 are also volatile. When AMX support is present, the TMM tile registers are volatile. Consider volatile registers destroyed on function calls unless otherwise safety-provable by analysis such as whole program optimization.

The x64 ABI considers registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, R15, and XMM6-XMM15 nonvolatile. They must be saved and restored by a function that uses them.

Function pointers

Function pointers are simply pointers to the label of the respective function. There's no table of contents (TOC) requirement for function pointers.

Floating-point support for older code

The MMX and floating-point stack registers (MM0-MM7/ST0-ST7) are preserved across context switches. There's no explicit calling convention for these registers. The use of these registers is strictly prohibited in kernel mode code.

FPCSR

The register state also includes the x87 FPU control word. The calling convention dictates this register to be nonvolatile.

The x87 FPU control word register gets set using the following standard values at the start of program execution:

Register[bits]	Setting
FPCSR[0:6]	Exception masks all 1's (all exceptions masked)
FPCSR[7]	Reserved - 0
FPCSR[8:9]	Precision Control - 10B (double precision)
FPCSR[10:11]	Rounding control - 0 (round to nearest)
FPCSR[12]	Infinity control - 0 (not used)

A callee that modifies any of the fields within FPCSR must restore them before returning to its caller. Furthermore, a caller that has modified any of these fields must restore them to their standard values before invoking a callee, unless by agreement the callee expects the modified values.

There are two exceptions to the rules about the non-volatility of the control flags:

- In functions where the documented purpose of the given function is to modify the nonvolatile FPCSR flags.
- When it's provably correct that the violation of these rules results in a program that behaves the same as a program that doesn't violate the rules, for example, through whole-program analysis.

MXCSR

The register state also includes MXCSR. The calling convention divides this register into a volatile portion and a nonvolatile portion. The volatile portion consists of the six status

flags, in MXCSR[0:5], while the rest of the register, MXCSR[6:15], is considered nonvolatile.

The nonvolatile portion is set to the following standard values at the start of program execution:

Register[bits]	Setting
MXCSR[6]	Denormals are zeros - 0
MXCSR[7:12]	Exception masks all 1's (all exceptions masked)
MXCSR[13:14]	Rounding control - 0 (round to nearest)
MXCSR[15]	Flush to zero for masked underflow - 0 (off)

A callee that modifies any of the nonvolatile fields within MXCSR must restore them before returning to its caller. Furthermore, a caller that has modified any of these fields must restore them to their standard values before invoking a callee, unless by agreement the callee expects the modified values.

There are two exceptions to the rules about the non-volatility of the control flags:

- In functions where the documented purpose of the given function is to modify the nonvolatile MXCSR flags.
- When it's provably correct that the violation of these rules results in a program that behaves the same as a program that doesn't violate the rules, for example, through whole-program analysis.

Make no assumptions about the MXCSR register's volatile portion state across a function boundary, unless the function documentation explicitly describes it.

setjmp/longjmp

When you include setjmpex.h or setjmp.h, all calls to setjmp or longjmp result in an unwind that invokes destructors and `__finally` calls. This behavior differs from x86, where including setjmp.h results in `__finally` clauses and destructors not being invoked.

A call to `setjmp` preserves the current stack pointer, non-volatile registers, and MXCSR registers. Calls to `longjmp` return to the most recent `setjmp` call site and resets the stack pointer, non-volatile registers, and MXCSR registers, back to the state as preserved by the most recent `setjmp` call.

See also

x64 software conventions

x64 stack usage

Article • 08/03/2021

All memory beyond the current address of RSP is considered volatile: The OS, or a debugger, may overwrite this memory during a user debug session, or an interrupt handler. Thus, RSP must always be set before attempting to read or write values to a stack frame.

This section discusses the allocation of stack space for local variables and the **alloca** intrinsic.

Stack allocation

A function's prolog is responsible for allocating stack space for local variables, saved registers, stack parameters, and register parameters.

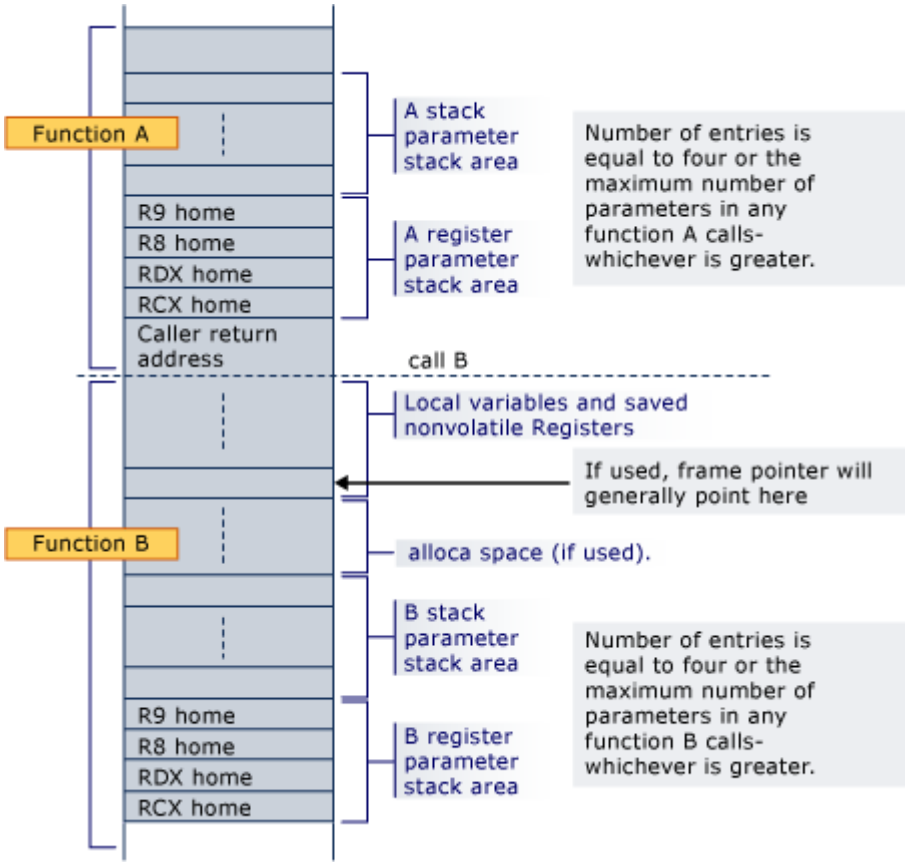
The parameter area is always at the bottom of the stack (even if `alloca` is used), so that it will always be adjacent to the return address during any function call. It contains at least four entries, but always enough space to hold all the parameters needed by any function that may be called. Note that space is always allocated for the register parameters, even if the parameters themselves are never homed to the stack; a callee is guaranteed that space has been allocated for all its parameters. Home addresses are required for the register arguments so a contiguous area is available in case the called function needs to take the address of the argument list (va_list) or an individual argument. This area also provides a convenient place to save register arguments during thunk execution and as a debugging option (for example, it makes the arguments easy to find during debugging if they are stored at their home addresses in the prolog code). Even if the called function has fewer than 4 parameters, these 4 stack locations are effectively owned by the called function, and may be used by the called function for other purposes besides saving parameter register values. Thus the caller may not save information in this region of stack across a function call.

If space is dynamically allocated (`alloca`) in a function, then a nonvolatile register must be used as a frame pointer to mark the base of the fixed part of the stack and that register must be saved and initialized in the prolog. Note that when `alloca` is used, calls to the same callee from the same caller may have different home addresses for their register parameters.

The stack will always be maintained 16-byte aligned, except within the prolog (for example, after the return address is pushed), and except where indicated in Function

Types for a certain class of frame functions.

The following is an example of the stack layout where function A calls a non-leaf function B. Function A's prolog has already allocated space for all the register and stack parameters required by B at the bottom of the stack. The call pushes the return address and B's prolog allocates space for its local variables, nonvolatile registers, and the space needed for it to call functions. If B uses `alloca`, the space is allocated between the local variable/nonvolatile register save area and the parameter stack area.



When the function B calls another function, the return address is pushed just below the home address for RCX.

Dynamic parameter stack area construction

If a frame pointer is used, the option exists to dynamically create the parameter stack area. This is not currently done in the x64 compiler.

Function types

There are basically two types of functions. A function that requires a stack frame is called a *frame function*. A function that does not require a stack frame is called a *leaf function*.

A frame function is a function that allocates stack space, calls other functions, saves nonvolatile registers, or uses exception handling. It also requires a function table entry. A frame function requires a prolog and an epilog. A frame function can dynamically allocate stack space and can employ a frame pointer. A frame function has the full capabilities of this calling standard at its disposal.

If a frame function does not call another function then it is not required to align the stack (referenced in Section Stack Allocation).

A leaf function is one that does not require a function table entry. It can't make changes to any nonvolatile registers, including RSP, which means that it can't call any functions or allocate stack space. It is allowed to leave the stack unaligned while it executes.

malloc alignment

malloc is guaranteed to return memory that's suitably aligned for storing any object that has a fundamental alignment and that could fit in the amount of memory that's allocated. A *fundamental alignment* is an alignment that's less than or equal to the largest alignment that's supported by the implementation without an alignment specification. (In Visual C++, this is the alignment that's required for a `double`, or 8 bytes. In code that targets 64-bit platforms, it's 16 bytes.) For example, a four-byte allocation would be aligned on a boundary that supports any four-byte or smaller object.

Visual C++ permits types that have *extended alignment*, which are also known as *over-aligned* types. For example, the SSE types __m128 and `__m256`, and types that are declared by using `__declspec(align( n ))` where `n` is greater than 8, have extended alignment. Memory alignment on a boundary that's suitable for an object that requires extended alignment is not guaranteed by `malloc`. To allocate memory for over-aligned types, use _aligned_malloc and related functions.

alloca

_alloca is required to be 16-byte aligned and additionally required to use a frame pointer.

The stack that is allocated needs to include space after it for parameters of subsequently called functions, as discussed in Stack Allocation.

See also

x64 software conventions

align

__declspec

x64 prolog and epilog

Article • 08/03/2021

Every function that allocates stack space, calls other functions, saves nonvolatile registers, or uses exception handling must have a prolog whose address limits are described in the unwind data associated with the respective function table entry. For more information, see x64 exception handling. The prolog saves argument registers in their home addresses if necessary, pushes nonvolatile registers on the stack, allocates the fixed part of the stack for locals and temporaries, and optionally establishes a frame pointer. The associated unwind data must describe the action of the prolog and must provide the information necessary to undo the effect of the prolog code.

If the fixed allocation in the stack is more than one page (that is, greater than 4096 bytes), then it's possible that the stack allocation could span more than one virtual memory page and, therefore, the allocation must be checked before it's allocated. A special routine that's callable from the prolog and which doesn't destroy any of the argument registers is provided for this purpose.

The preferred method for saving nonvolatile registers is to move them onto the stack before the fixed stack allocation. If the fixed stack allocation is performed before the nonvolatile registers are saved, then most probably a 32-bit displacement is required to address the saved register area. (Reportedly, pushes of registers are as fast as moves and should remain so for the foreseeable future in spite of the implied dependency between pushes.) Nonvolatile registers can be saved in any order. However, the first use of a nonvolatile register in the prolog must be to save it.

Prolog code

The code for a typical prolog might be:

MASM

```
    mov    [RSP + 8], RCX
    push   R15
    push   R14
    push   R13
    sub    RSP, fixed-allocation-size
    lea    R13, 128[RSP]
    ...
```

This prolog stores the argument register RCX in its home location, saves nonvolatile registers R13-R15, allocates the fixed part of the stack frame, and establishes a frame

pointer that points 128 bytes into the fixed allocation area. Using an offset allows more of the fixed allocation area to be addressed with one-byte offsets.

If the fixed allocation size is greater than or equal to one page of memory, then a helper function must be called before modifying RSP. This helper, `__chkstk`, probes the to-be-allocated stack range to ensure that the stack is extended properly. In that case, the previous prolog example would instead be:

MASM

```
    mov    [RSP + 8], RCX
    push   R15
    push   R14
    push   R13
    mov    RAX,  fixed-allocation-size
    call   __chkstk
    sub    RSP, RAX
    lea    R13, 128[RSP]
    ...
```

The `__chkstk` helper will not modify any registers other than R10, R11, and the condition codes. In particular, it will return RAX unchanged and leave all nonvolatile registers and argument-passing registers unmodified.

Epilog code

Epilog code exists at each exit to a function. Whereas there is normally only one prolog, there can be many epilogs. Epilog code trims the stack to its fixed allocation size (if necessary), deallocates the fixed stack allocation, restores nonvolatile registers by popping their saved values from the stack, and returns.

The epilog code must follow a strict set of rules for the unwind code to reliably unwind through exceptions and interrupts. These rules reduce the amount of unwind data required, because no extra data is needed to describe each epilog. Instead, the unwind code can determine that an epilog is being executed by scanning forward through a code stream to identify an epilog.

If no frame pointer is used in the function, then the epilog must first deallocate the fixed part of the stack, the nonvolatile registers are popped, and control is returned to the calling function. For example,

MASM

```
    add      RSP, fixed-allocation-size
    pop      R13
```

```
    pop     R14
    pop     R15
    ret
```

If a frame pointer is used in the function, then the stack must be trimmed to its fixed allocation prior to the execution of the epilog. This action is technically not part of the epilog. For example, the following epilog could be used to undo the prolog previously used:

MASM

```
    lea     RSP, -128[R13]
    ; epilogue proper starts here
    add     RSP, fixed-allocation-size
    pop     R13
    pop     R14
    pop     R15
    ret
```

In practice, when a frame pointer is used, there is no good reason to adjust RSP in two steps, so the following epilog would be used instead:

MASM

```
    lea     RSP, fixed-allocation-size - 128[R13]
    pop     R13
    pop     R14
    pop     R15
    ret
```

These forms are the only legal ones for an epilog. It must consist of either an `add RSP,constant` or `lea RSP,constant[FPReg]`, followed by a series of zero or more 8-byte register pops and a `return` or a `jmp`. (Only a subset of `jmp` statements are allowable in the epilog. The subset is exclusively the class of `jmp` statements with ModRM memory references where ModRM mod field value is 00. The use of `jmp` statements in the epilog with ModRM mod field value 01 or 10 is prohibited. See Table A-15 in the AMD x86-64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions, for more info on the allowable ModRM references.) No other code can appear. In particular, nothing can be scheduled within an epilog, including loading of a return value.

When a frame pointer is not used, the epilog must use `add RSP,constant` to deallocate the fixed part of the stack. It may not use `lea RSP,constant[RSP]` instead. This restriction exists so the unwind code has fewer patterns to recognize when searching for epilogs.

Following these rules allows the unwind code to determine that an epilog is currently being executed and to simulate execution of the remainder of the epilog to allow recreating the context of the calling function.

See also

x64 Software Conventions

x64 exception handling

Article • 02/08/2022

An overview of structured exception handling and C++ exception handling coding conventions and behavior on the x64. For general information on exception handling, see Exception Handling in Visual C++.

Unwind data for exception handling, debugger support

Several data structures are required for exception handling and debugging support.

struct RUNTIME_FUNCTION

Table-based exception handling requires a table entry for all functions that allocate stack space or call another function (for example, nonleaf functions). Function table entries have the format:

Size	Value
ULONG	Function start address
ULONG	Function end address
ULONG	Unwind info address

The RUNTIME_FUNCTION structure must be DWORD aligned in memory. All addresses are image relative, that is, they're 32-bit offsets from the starting address of the image that contains the function table entry. These entries are sorted, and put in the .pdata section of a PE32+ image. For dynamically generated functions [JIT compilers], the runtime to support these functions must either use RtlInstallFunctionTableCallback or RtlAddFunctionTable to provide this information to the operating system. Failure to do so will result in unreliable exception handling and debugging of processes.

struct UNWIND_INFO

The unwind data info structure is used to record the effects a function has on the stack pointer, and where the nonvolatile registers are saved on the stack:

Size	Value

Size	Value
UBYTE: 3	Version
UBYTE: 5	Flags
UBYTE	Size of prolog
UBYTE	Count of unwind codes
UBYTE: 4	Frame Register
UBYTE: 4	Frame Register offset (scaled)
USHORT * n	Unwind codes array
variable	Can either be of form (1) or (2) below

(1) Exception Handler

Size	Value
ULONG	Address of exception handler
variable	Language-specific handler data (optional)

(2) Chained Unwind Info

Size	Value
ULONG	Function start address
ULONG	Function end address
ULONG	Unwind info address

The UNWIND_INFO structure must be DWORD aligned in memory. Here's what each field means:

- **Version**

 Version number of the unwind data, currently 1.

- **Flags**

 Three flags are currently defined:

Flag	Description

Flag	Description
`UNW_FLAG_EHANDLER`	The function has an exception handler that should be called when looking for functions that need to examine exceptions.
`UNW_FLAG_UHANDLER`	The function has a termination handler that should be called when unwinding an exception.
`UNW_FLAG_CHAININFO`	This unwind info structure is not the primary one for the procedure. Instead, the chained unwind info entry is the contents of a previous RUNTIME_FUNCTION entry. For information, see Chained unwind info structures. If this flag is set, then the UNW_FLAG_EHANDLER and UNW_FLAG_UHANDLER flags must be cleared. Also, the frame register and fixed-stack allocation fields must have the same values as in the primary unwind info.

- **Size of prolog**

 Length of the function prolog in bytes.

- **Count of unwind codes**

 The number of slots in the unwind codes array. Some unwind codes, for example, UWOP_SAVE_NONVOL, require more than one slot in the array.

- **Frame register**

 If nonzero, then the function uses a frame pointer (FP), and this field is the number of the nonvolatile register used as the frame pointer, using the same encoding for the operation info field of UNWIND_CODE nodes.

- **Frame register offset (scaled)**

 If the frame register field is nonzero, this field is the scaled offset from RSP that is applied to the FP register when it's established. The actual FP register is set to RSP + 16 * this number, allowing offsets from 0 to 240. This offset permits pointing the FP register into the middle of the local stack allocation for dynamic stack frames, allowing better code density through shorter instructions. (That is, more instructions can use the 8-bit signed offset form.)

- **Unwind codes array**

 An array of items that explains the effect of the prolog on the nonvolatile registers and RSP. See the section on UNWIND_CODE for the meanings of individual items. For alignment purposes, this array always has an even number of entries, and the final entry is potentially unused. In that case, the array is one longer than indicated by the count of unwind codes field.

- **Address of exception handler**

 An image-relative pointer to either the function's language-specific exception or termination handler, if flag UNW_FLAG_CHAININFO is clear and one of the flags UNW_FLAG_EHANDLER or UNW_FLAG_UHANDLER is set.

- **Language-specific handler data**

 The function's language-specific exception handler data. The format of this data is unspecified and completely determined by the specific exception handler in use.

- **Chained Unwind Info**

 If flag UNW_FLAG_CHAININFO is set, then the UNWIND_INFO structure ends with three UWORDs. These UWORDs represent the RUNTIME_FUNCTION information for the function of the chained unwind.

struct UNWIND_CODE

The unwind code array is used to record the sequence of operations in the prolog that affect the nonvolatile registers and RSP. Each code item has this format:

Size	Value
UBYTE	Offset in prolog
UBYTE: 4	Unwind operation code
UBYTE: 4	Operation info

The array is sorted by descending order of offset in the prolog.

Offset in prolog

Offset (from the beginning of the prolog) of the end of the instruction that performs this operation, plus 1 (that is, the offset of the start of the next instruction).

Unwind operation code

Note: Certain operation codes require an unsigned offset to a value in the local stack frame. This offset is from the start, that is, the lowest address of the fixed stack allocation. If the Frame Register field in the UNWIND_INFO is zero, this offset is from RSP. If the Frame Register field is nonzero, this offset is from where RSP was located when the FP register was established. It equals the FP register minus the FP register

offset (16 * the scaled frame register offset in the UNWIND_INFO). If an FP register is used, then any unwind code taking an offset must only be used after the FP register is established in the prolog.

For all opcodes except `UWOP_SAVE_XMM128` and `UWOP_SAVE_XMM128_FAR`, the offset is always a multiple of 8, because all stack values of interest are stored on 8-byte boundaries (the stack itself is always 16-byte aligned). For operation codes that take a short offset (less than 512K), the final USHORT in the nodes for this code holds the offset divided by 8. For operation codes that take a long offset (512K <= offset < 4GB), the final two USHORT nodes for this code hold the offset (in little-endian format).

For the opcodes `UWOP_SAVE_XMM128` and `UWOP_SAVE_XMM128_FAR`, the offset is always a multiple of 16, since all 128-bit XMM operations must occur on 16-byte aligned memory. Therefore, a scale factor of 16 is used for `UWOP_SAVE_XMM128`, permitting offsets of less than 1M.

The unwind operation code is one of these values:

- `UWOP_PUSH_NONVOL` (0) 1 node

 Push a nonvolatile integer register, decrementing RSP by 8. The operation info is the number of the register. Because of the constraints on epilogs, `UWOP_PUSH_NONVOL` unwind codes must appear first in the prolog and correspondingly, last in the unwind code array. This relative ordering applies to all other unwind codes except `UWOP_PUSH_MACHFRAME`.

- `UWOP_ALLOC_LARGE` (1) 2 or 3 nodes

 Allocate a large-sized area on the stack. There are two forms. If the operation info equals 0, then the size of the allocation divided by 8 is recorded in the next slot, allowing an allocation up to 512K - 8. If the operation info equals 1, then the unscaled size of the allocation is recorded in the next two slots in little-endian format, allowing allocations up to 4GB - 8.

- `UWOP_ALLOC_SMALL` (2) 1 node

 Allocate a small-sized area on the stack. The size of the allocation is the operation info field * 8 + 8, allowing allocations from 8 to 128 bytes.

 The unwind code for a stack allocation should always use the shortest possible encoding:

Allocation Size	Unwind Code

Allocation Size	Unwind Code
8 to 128 bytes	`UWOP_ALLOC_SMALL`
136 to 512K-8 bytes	`UWOP_ALLOC_LARGE`, operation info = 0
512K to 4G-8 bytes	`UWOP_ALLOC_LARGE`, operation info = 1

- `UWOP_SET_FPREG` (3) 1 node

 Establish the frame pointer register by setting the register to some offset of the current RSP. The offset is equal to the Frame Register offset (scaled) field in the UNWIND_INFO * 16, allowing offsets from 0 to 240. The use of an offset permits establishing a frame pointer that points to the middle of the fixed stack allocation, helping code density by allowing more accesses to use short instruction forms. The operation info field is reserved and shouldn't be used.

- `UWOP_SAVE_NONVOL` (4) 2 nodes

 Save a nonvolatile integer register on the stack using a MOV instead of a PUSH. This code is primarily used for *shrink-wrapping*, where a nonvolatile register is saved to the stack in a position that was previously allocated. The operation info is the number of the register. The scaled-by-8 stack offset is recorded in the next unwind operation code slot, as described in the note above.

- `UWOP_SAVE_NONVOL_FAR` (5) 3 nodes

 Save a nonvolatile integer register on the stack with a long offset, using a MOV instead of a PUSH. This code is primarily used for *shrink-wrapping*, where a nonvolatile register is saved to the stack in a position that was previously allocated. The operation info is the number of the register. The unscaled stack offset is recorded in the next two unwind operation code slots, as described in the note above.

- `UWOP_SAVE_XMM128` (8) 2 nodes

 Save all 128 bits of a nonvolatile XMM register on the stack. The operation info is the number of the register. The scaled-by-16 stack offset is recorded in the next slot.

- `UWOP_SAVE_XMM128_FAR` (9) 3 nodes

 Save all 128 bits of a nonvolatile XMM register on the stack with a long offset. The operation info is the number of the register. The unscaled stack offset is recorded in the next two slots.

- `UWOP_PUSH_MACHFRAME` (10) 1 node

 Push a machine frame. This unwind code is used to record the effect of a hardware interrupt or exception. There are two forms. If the operation info equals 0, one of these frames has been pushed on the stack:

Location	Value
RSP+32	SS
RSP+24	Old RSP
RSP+16	EFLAGS
RSP+8	CS
RSP	RIP

 If the operation info equals 1, then one of these frames has been pushed:

Location	Value
RSP+40	SS
RSP+32	Old RSP
RSP+24	EFLAGS
RSP+16	CS
RSP+8	RIP
RSP	Error code

 This unwind code always appears in a dummy prolog, which is never actually executed, but instead appears before the real entry point of an interrupt routine, and exists only to provide a place to simulate the push of a machine frame. `UWOP_PUSH_MACHFRAME` records that simulation, which indicates the machine has conceptually done this operation:

 1. Pop RIP return address from top of stack into *Temp*
 2. Push SS
 3. Push old RSP
 4. Push EFLAGS
 5. Push CS

6. Push *Temp*

7. Push Error Code (if op info equals 1)

The simulated `UWOP_PUSH_MACHFRAME` operation decrements RSP by 40 (op info equals 0) or 48 (op info equals 1).

Operation info

The meaning of the operation info bits depends upon the operation code. To encode a general-purpose (integer) register, this mapping is used:

Bit	Register
0	RAX
1	RCX
2	RDX
3	RBX
4	RSP
5	RBP
6	RSI
7	RDI
8 to 15	R8 to R15

Chained unwind info structures

If the UNW_FLAG_CHAININFO flag is set, then an unwind info structure is a secondary one, and the shared exception-handler/chained-info address field contains the primary unwind information. This sample code retrieves the primary unwind information, assuming that `unwindInfo` is the structure that has the UNW_FLAG_CHAININFO flag set.

```
C++

PRUNTIME_FUNCTION primaryUwindInfo = (PRUNTIME_FUNCTION)&(unwindInfo-
>UnwindCode[( unwindInfo->CountOfCodes + 1 ) & ~1]);
```

Chained info is useful in two situations. First, it can be used for noncontiguous code segments. By using chained info, you can reduce the size of the required unwind

information, because you do not have to duplicate the unwind codes array from the primary unwind info.

You can also use chained info to group volatile register saves. The compiler may delay saving some volatile registers until it is outside of the function entry prolog. You can record them by having primary unwind info for the portion of the function before the grouped code, and then setting up chained info with a non-zero size of prolog, where the unwind codes in the chained info reflect saves of the nonvolatile registers. In that case, the unwind codes are all instances of UWOP_SAVE_NONVOL. A grouping that saves nonvolatile registers by using a PUSH or modifies the RSP register by using an additional fixed stack allocation is not supported.

An UNWIND_INFO item that has UNW_FLAG_CHAININFO set can contain a RUNTIME_FUNCTION entry whose UNWIND_INFO item also has UNW_FLAG_CHAININFO set, sometimes called *multiple shrink-wrapping*. Eventually, the chained unwind info pointers arrive at an UNWIND_INFO item that has UNW_FLAG_CHAININFO cleared. This item is the primary UNWIND_INFO item, which points to the actual procedure entry point.

Unwind procedure

The unwind code array is sorted into descending order. When an exception occurs, the complete context is stored by the operating system in a context record. The exception dispatch logic is then invoked, which repeatedly executes these steps to find an exception handler:

1. Use the current RIP stored in the context record to search for a RUNTIME_FUNCTION table entry that describes the current function (or function portion, for chained UNWIND_INFO entries).

2. If no function table entry is found, then it's in a leaf function, and RSP directly addresses the return pointer. The return pointer at [RSP] is stored in the updated context, the simulated RSP is incremented by 8, and step 1 is repeated.

3. If a function table entry is found, RIP can lie within three regions: a) in an epilog, b) in the prolog, or c) in code that may be covered by an exception handler.

 - Case a) If the RIP is within an epilog, then control is leaving the function, there can be no exception handler associated with this exception for this function, and the effects of the epilog must be continued to compute the context of the caller function. To determine if the RIP is within an epilog, the code stream from RIP onward is examined. If that code stream can be

matched to the trailing portion of a legitimate epilog, then it's in an epilog, and the remaining portion of the epilog is simulated, with the context record updated as each instruction is processed. After this processing, step 1 is repeated.

- Case b) If the RIP lies within the prologue, then control hasn't entered the function, there can be no exception handler associated with this exception for this function, and the effects of the prolog must be undone to compute the context of the caller function. The RIP is within the prolog if the distance from the function start to the RIP is less than or equal to the prolog size encoded in the unwind info. The effects of the prolog are unwound by scanning forward through the unwind codes array for the first entry with an offset less than or equal to the offset of the RIP from the function start, then undoing the effect of all remaining items in the unwind code array. Step 1 is then repeated.

- Case c) If the RIP isn't within a prolog or epilog, and the function has an exception handler (UNW_FLAG_EHANDLER is set), then the language-specific handler is called. The handler scans its data and calls filter functions as appropriate. The language-specific handler can return that the exception was handled or that the search is to be continued. It can also initiate an unwind directly.

4. If the language-specific handler returns a handled status, then execution is continued using the original context record.

5. If there's no language-specific handler or the handler returns a "continue search" status, then the context record must be unwound to the state of the caller. It's done by processing all of the unwind code array elements, undoing the effect of each. Step 1 is then repeated.

When chained unwind info is involved, these basic steps are still followed. The only difference is that, while walking the unwind code array to unwind a prolog's effects, once the end of the array is reached, it's then linked to the parent unwind info and the entire unwind code array found there is walked. This linking continues until arriving at an unwind info without the UNW_CHAINED_INFO flag, and then it finishes walking its unwind code array.

The smallest set of unwind data is 8 bytes. This would represent a function that only allocated 128 bytes of stack or less, and possibly saved one nonvolatile register. It's also the size of a chained unwind info structure for a zero-length prolog with no unwind codes.

Language-specific handler

The relative address of the language-specific handler is present in the UNWIND_INFO whenever flags UNW_FLAG_EHANDLER or UNW_FLAG_UHANDLER are set. As described in the previous section, the language-specific handler is called as part of the search for an exception handler or as part of an unwind. It has this prototype:

```
C++

typedef EXCEPTION_DISPOSITION (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN ULONG64 EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);
```

ExceptionRecord supplies a pointer to an exception record, which has the standard Win64 definition.

EstablisherFrame is the address of the base of the fixed stack allocation for this function.

ContextRecord points to the exception context at the time the exception was raised (in the exception handler case) or the current "unwind" context (in the termination handler case).

DispatcherContext points to the dispatcher context for this function. It has this definition:

```
C++

typedef struct _DISPATCHER_CONTEXT {
    ULONG64 ControlPc;
    ULONG64 ImageBase;
    PRUNTIME_FUNCTION FunctionEntry;
    ULONG64 EstablisherFrame;
    ULONG64 TargetIp;
    PCONTEXT ContextRecord;
    PEXCEPTION_ROUTINE LanguageHandler;
    PVOID HandlerData;
} DISPATCHER_CONTEXT, *PDISPATCHER_CONTEXT;
```

ControlPc is the value of RIP within this function. This value is either an exception address or the address at which control left the establishing function. The RIP is used to determine if control is within some guarded construct inside this function, for example, a `__try` block for `__try`/`__except` or `__try`/`__finally`.

ImageBase is the image base (load address) of the module containing this function, to be added to the 32-bit offsets used in the function entry and unwind info to record relative addresses.

FunctionEntry supplies a pointer to the RUNTIME_FUNCTION function entry holding the function and unwind info image-base relative addresses for this function.

EstablisherFrame is the address of the base of the fixed stack allocation for this function.

TargetIp Supplies an optional instruction address that specifies the continuation address of the unwind. This address is ignored if **EstablisherFrame** isn't specified.

ContextRecord points to the exception context, for use by the system exception dispatch/unwind code.

LanguageHandler points to the language-specific language handler routine being called.

HandlerData points to the language-specific handler data for this function.

Unwind helpers for MASM

In order to write proper assembly routines, there's a set of pseudo-operations that can be used in parallel with the actual assembly instructions to create the appropriate .pdata and .xdata. And, there's a set of macros that provide simplified use of the pseudo-operations for their most common uses.

Raw pseudo-operations

Pseudo operation	Description
PROC FRAME [:*ehandler*]	Causes MASM to generate a function table entry in .pdata and unwind information in .xdata for a function's structured exception handling unwind behavior. If *ehandler* is present, this proc is entered in the .xdata as the language-specific handler. When the FRAME attribute is used, it must be followed by an .ENDPROLOG directive. If the function is a leaf function (as defined in Function types) the FRAME attribute is unnecessary, as are the remainder of these pseudo-operations.

Pseudo operation	Description
.PUSHREG *register*	Generates a UWOP_PUSH_NONVOL unwind code entry for the specified register number using the current offset in the prologue. Only use it with nonvolatile integer registers. For pushes of volatile registers, use an .ALLOCSTACK 8, instead
.SETFRAME *register*, *offset*	Fills in the frame register field and offset in the unwind information using the specified register and offset. The offset must be a multiple of 16 and less than or equal to 240. This directive also generates a UWOP_SET_FPREG unwind code entry for the specified register using the current prologue offset.
.ALLOCSTACK *size*	Generates a UWOP_ALLOC_SMALL or a UWOP_ALLOC_LARGE with the specified size for the current offset in the prologue. The *size* operand must be a multiple of 8.
.SAVEREG *register*, *offset*	Generates either a UWOP_SAVE_NONVOL or a UWOP_SAVE_NONVOL_FAR unwind code entry for the specified register and offset using the current prologue offset. MASM chooses the most efficient encoding. *offset* must be positive, and a multiple of 8. *offset* is relative to the base of the procedure's frame, which is generally in RSP, or, if using a frame pointer, the unscaled frame pointer.
.SAVEXMM128 *register*, *offset*	Generates either a UWOP_SAVE_XMM128 or a UWOP_SAVE_XMM128_FAR unwind code entry for the specified XMM register and offset using the current prologue offset. MASM chooses the most efficient encoding. *offset* must be positive, and a multiple of 16. *offset* is relative to the base of the procedure's frame, which is generally in RSP, or, if using a frame pointer, the unscaled frame pointer.
.PUSHFRAME [*code*]	Generates a UWOP_PUSH_MACHFRAME unwind code entry. If the optional *code* is specified, the unwind code entry is given a modifier of 1. Otherwise the modifier is 0.
.ENDPROLOG	Signals the end of the prologue declarations. Must occur in the first 255 bytes of the function.

Here's a sample function prolog with proper usage of most of the opcodes:

MASM

```
sample PROC FRAME
    db      048h; emit a REX prefix, to enable hot-patching
    push rbp
    .pushreg rbp
```

```
    sub rsp, 040h
    .allocstack 040h
    lea rbp, [rsp+020h]
    .setframe rbp, 020h
    movdqa [rbp], xmm7
    .savexmm128 xmm7, 020h ;the offset is from the base of the frame
                           ;not the scaled offset of the frame
    mov [rbp+018h], rsi
    .savereg rsi, 038h
    mov [rsp+010h], rdi
    .savereg rdi, 010h ; you can still use RSP as the base of the frame
                       ; or any other register you choose
    .endprolog

; you can modify the stack pointer outside of the prologue (similar to
alloca)
; because we have a frame pointer.
; if we didn't have a frame pointer, this would be illegal
; if we didn't make this modification,
; there would be no need for a frame pointer

    sub rsp, 060h

; we can unwind from the next AV because of the frame pointer

    mov rax, 0
    mov rax, [rax] ; AV!

; restore the registers that weren't saved with a push
; this isn't part of the official epilog, as described in section 2.5

    movdqa xmm7, [rbp]
    mov rsi, [rbp+018h]
    mov rdi, [rbp-010h]

; Here's the official epilog

    lea rsp, [rbp+020h] ; deallocate both fixed and dynamic portions of the
frame
    pop rbp
    ret
sample ENDP
```

For more information about the epilog example, see Epilog code in x64 prolog and epilog.

MASM macros

In order to simplify the use of the Raw pseudo-operations, there's a set of macros, defined in ksamd64.inc, which can be used to create typical procedure prologues and epilogues.

Macro	Description
alloc_stack(n)	Allocates a stack frame of n bytes (using `sub rsp, n`), and emits the appropriate unwind information (.allocstack n)
save_reg *reg*, *loc*	Saves a nonvolatile register *reg* on the stack at RSP offset *loc*, and emits the appropriate unwind information. (.savereg reg, loc)
push_reg *reg*	Pushes a nonvolatile register *reg* on the stack, and emits the appropriate unwind information. (.pushreg reg)
rex_push_reg *reg*	Saves a nonvolatile register on the stack using a 2-byte push, and emits the appropriate unwind information (.pushreg reg). Use this macro if the push is the first instruction in the function, to ensure that the function is hot-patchable.
save_xmm128 *reg*, *loc*	Saves a nonvolatile XMM register *reg* on the stack at RSP offset *loc*, and emits the appropriate unwind information (.savexmm128 reg, loc)
set_frame *reg*, *offset*	Sets the frame register *reg* to be the RSP + *offset* (using a `mov`, or an `lea`), and emits the appropriate unwind information (.set_frame reg, offset)
push_eflags	Pushes the eflags with a `pushfq` instruction, and emits the appropriate unwind information (.alloc_stack 8)

Here's a sample function prolog with proper usage of the macros:

MASM

```
sampleFrame struct
    Fill     dq ?; fill to 8 mod 16
    SavedRdi dq ?; Saved Register RDI
    SavedRsi dq ?; Saved Register RSI
sampleFrame ends

sample2 PROC FRAME
    alloc_stack(sizeof sampleFrame)
    save_reg rdi, sampleFrame.SavedRdi
    save_reg rsi, sampleFrame.SavedRsi
    .end_prolog

; function body

    mov rsi, sampleFrame.SavedRsi[rsp]
    mov rdi, sampleFrame.SavedRdi[rsp]

; Here's the official epilog

    add rsp, (sizeof sampleFrame)
    ret
sample2 ENDP
```

Unwind data definitions in C

Here's a C description of the unwind data:

C

```c
typedef enum _UNWIND_OP_CODES {
    UWOP_PUSH_NONVOL = 0, /* info == register number */
    UWOP_ALLOC_LARGE,     /* no info, alloc size in next 2 slots */
    UWOP_ALLOC_SMALL,     /* info == size of allocation / 8 - 1 */
    UWOP_SET_FPREG,       /* no info, FP = RSP + UNWIND_INFO.FPRegOffset*16
*/
    UWOP_SAVE_NONVOL,     /* info == register number, offset in next slot */
    UWOP_SAVE_NONVOL_FAR, /* info == register number, offset in next 2 slots
*/
    UWOP_SAVE_XMM128 = 8, /* info == XMM reg number, offset in next slot */
    UWOP_SAVE_XMM128_FAR, /* info == XMM reg number, offset in next 2 slots
*/
    UWOP_PUSH_MACHFRAME   /* info == 0: no error-code, 1: error-code */
} UNWIND_CODE_OPS;

typedef unsigned char UBYTE;

typedef union _UNWIND_CODE {
    struct {
        UBYTE CodeOffset;
        UBYTE UnwindOp : 4;
        UBYTE OpInfo   : 4;
    };
    USHORT FrameOffset;
} UNWIND_CODE, *PUNWIND_CODE;

#define UNW_FLAG_EHANDLER  0x01
#define UNW_FLAG_UHANDLER  0x02
#define UNW_FLAG_CHAININFO 0x04

typedef struct _UNWIND_INFO {
    UBYTE Version       : 3;
    UBYTE Flags         : 5;
    UBYTE SizeOfProlog;
    UBYTE CountOfCodes;
    UBYTE FrameRegister : 4;
    UBYTE FrameOffset   : 4;
    UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes + 1) & ~1) - 1];
*   union {
*       OPTIONAL ULONG ExceptionHandler;
*       OPTIONAL ULONG FunctionEntry;
*   };
*   OPTIONAL ULONG ExceptionData[]; */
} UNWIND_INFO, *PUNWIND_INFO;

typedef struct _RUNTIME_FUNCTION {
```

```
    ULONG BeginAddress;
    ULONG EndAddress;
    ULONG UnwindData;
} RUNTIME_FUNCTION, *PRUNTIME_FUNCTION;

#define GetUnwindCodeEntry(info, index) \
    ((info)->UnwindCode[index])

#define GetLanguageSpecificDataPtr(info) \
    ((PVOID)&GetUnwindCodeEntry((info),((info)->CountOfCodes + 1) & ~1))

#define GetExceptionHandler(base, info) \
    ((PEXCEPTION_HANDLER)((base) + *
(PULONG)GetLanguageSpecificDataPtr(info)))

#define GetChainedFunctionEntry(base, info) \
    ((PRUNTIME_FUNCTION)((base) + *
(PULONG)GetLanguageSpecificDataPtr(info)))

#define GetExceptionDataPtr(info) \
    ((PVOID)((PULONG)GetLanguageSpecificData(info) + 1))
```

See also

x64 software conventions

Configure C++ projects for ARM processors

Article • 07/27/2023

This section of the documentation contains information about how to use the MSVC build tools to target ARM hardware.

In This Section

Overview of ARM ABI conventions
Describes the application binary interface used by Windows on ARM for register usage, calling conventions and exception handling.

Overview of ARM64 ABI conventions
Describes the application binary interface used by Windows on ARM64 for register usage, calling conventions and exception handling.

Common MSVC ARM migration issues
Describes C++ code elements that are commonly assumed to be portable across architectures, but that produce different results for ARM than for x86 and x64.

ARM exception handling
Describes the encoding scheme for stack unwinding during structured exception handling in Windows on ARM.

ARM64 exception handling
Describes the encoding scheme for stack unwinding during structured exception handling in Windows on ARM64.

Related Sections

Get started with Arm64EC
Describes how to get started building your app or project using Arm64EC.

How to: Configure projects to target platforms
Describes how to set up your build to target different processor architectures, including Arm64.

ARM intrinsics
Describes compiler intrinsics for processors that use the ARM architecture.

ARM64 intrinsics

Describes compiler intrinsics for processors that use the ARM64 architecture.

Common Visual C++ ARM Migration Issues

Article • 06/15/2022

When using the Microsoft C++ compiler (MSVC), the same C++ source code might produce different results on the ARM architecture than it does on x86 or x64 architectures.

Sources of migration issues

Many issues that you might encounter when you migrate code from the x86 or x64 architectures to the ARM architecture are related to source-code constructs that might invoke undefined, implementation-defined, or unspecified behavior.

Undefined behavior is behavior that the C++ standard does not define, and that's caused by an operation that has no reasonable result: for example, converting a floating-point value to an unsigned integer, or shifting a value by a number of positions that is negative or exceeds the number of bits in its promoted type.

Implementation-defined behavior is behavior that the C++ standard requires the compiler vendor to define and document. A program can safely rely on implementation-defined behavior, even though doing so might not be portable. Examples of implementation-defined behavior include the sizes of built-in data types and their alignment requirements. An example of an operation that might be affected by implementation-defined behavior is accessing the variable arguments list.

Unspecified behavior is behavior that the C++ standard leaves intentionally non-deterministic. Although the behavior is considered non-deterministic, particular invocations of unspecified behavior are determined by the compiler implementation. However, there is no requirement for a compiler vendor to predetermine the result or guarantee consistent behavior between comparable invocations, and there is no requirement for documentation. An example of unspecified behavior is the order in which sub-expressions, which include arguments to a function call, are evaluated.

Other migration issues can be attributed to hardware differences between ARM and x86 or x64 architectures that interact with the C++ standard differently. For example, the strong memory model of the x86 and x64 architecture gives `volatile`-qualified variables some additional properties that have been used to facilitate certain kinds of inter-thread communication in the past. But the ARM architecture's weak memory model doesn't support this use, nor does the C++ standard require it.

> **Important**
>
> Although `volatile` gains some properties that can be used to implement limited forms of inter-thread communication on x86 and x64, these additional properties are not sufficient to implement inter-thread communication in general. The C++ standard recommends that such communication be implemented by using appropriate synchronization primitives instead.

Because different platforms might express these kinds of behavior differently, porting software between platforms can be difficult and bug-prone if it depends on the behavior of a specific platform. Although many of these kinds of behavior can be observed and might appear stable, relying on them is at least non-portable, and in the cases of undefined or unspecified behavior, is also an error. Even the behavior that's cited in this document should not be relied on, and could change in future compilers or CPU implementations.

Example migration issues

The rest of this document describes how the different behavior of these C++ language elements can produce different results on different platforms.

Conversion of floating-point to unsigned integer

On the ARM architecture, conversion of a floating-point value to a 32-bit integer saturates to the nearest value that the integer can represent if the floating-point value is outside the range that the integer can represent. On the x86 and x64 architectures, the conversion wraps around if the integer is unsigned, or is set to -2147483648 if the integer is signed. None of these architectures directly support the conversion of floating-point values to smaller integer types; instead, the conversions are performed to 32 bits, and the results are truncated to a smaller size.

For the ARM architecture, the combination of saturation and truncation means that conversion to unsigned types correctly saturates smaller unsigned types when it saturates a 32-bit integer, but produces a truncated result for values that are larger than the smaller type can represent but too small to saturate the full 32-bit integer. Conversion also saturates correctly for 32-bit signed integers, but truncation of saturated, signed integers results in -1 for positively-saturated values and 0 for negatively-saturated values. Conversion to a smaller signed integer produces a truncated result that's unpredictable.

For the x86 and x64 architectures, the combination of wrap-around behavior for unsigned integer conversions and explicit valuation for signed integer conversions on overflow, together with truncation, make the results for most shifts unpredictable if they are too large.

These platforms also differ in how they handle conversion of NaN (Not-a-Number) to integer types. On ARM, NaN converts to 0x00000000; on x86 and x64, it converts to 0x80000000.

Floating-point conversion can only be relied on if you know that the value is within the range of the integer type that it's being converted to.

Shift operator (<< >>) behavior

On the ARM architecture, a value can be shifted left or right up to 255 bits before the pattern begins to repeat. On x86 and x64 architectures, the pattern is repeated at every multiple of 32 unless the source of the pattern is a 64-bit variable; in that case, the pattern repeats at every multiple of 64 on x64, and every multiple of 256 on x86, where a software implementation is employed. For example, for a 32-bit variable that has a value of 1 shifted left by 32 positions, on ARM the result is 0, on x86 the result is 1, and on x64 the result is also 1. However, if the source of the value is a 64-bit variable, then the result on all three platforms is 4294967296, and the value doesn't "wrap around" until it's shifted 64 positions on x64, or 256 positions on ARM and x86.

Because the result of a shift operation that exceeds the number of bits in the source type is undefined, the compiler is not required to have consistent behavior in all situations. For example, if both operands of a shift are known at compile time, the compiler may optimize the program by using an internal routine to precompute the result of the shift and then substituting the result in place of the shift operation. If the shift amount is too large, or negative, the result of the internal routine might be different than the result of the same shift expression as executed by the CPU.

Variable arguments (varargs) behavior

On the ARM architecture, parameters from the variable arguments list that are passed on the stack are subject to alignment. For example, a 64-bit parameter is aligned on a 64-bit boundary. On x86 and x64, arguments that are passed on the stack are not subject to alignment and pack tightly. This difference can cause a variadic function like `printf` to read memory addresses that were intended as padding on ARM if the expected layout of the variable arguments list is not matched exactly, even though it

might work for a subset of some values on the x86 or x64 architectures. Consider this example:

```C
// notice that a 64-bit integer is passed to the function, but '%d' is used
to read it.
// on x86 and x64 this may work for small values because %d will "parse" the
low-32 bits of the argument.
// on ARM the calling convention will align the 64-bit value and the code
will print a random value
printf("%d\n", 1LL);
```

In this case, the bug can be fixed by making sure that the correct format specification is used so that the alignment of the argument is considered. This code is correct:

```C
// CORRECT: use %I64d for 64-bit integers
printf("%I64d\n", 1LL);
```

Argument evaluation order

Because ARM, x86, and x64 processors are so different, they can present different requirements to compiler implementations, and also different opportunities for optimizations. Because of this, together with other factors like calling-convention and optimization settings, a compiler might evaluate function arguments in a different order on different architectures or when the other factors are changed. This can cause the behavior of an app that relies on a specific evaluation order to change unexpectedly.

This kind of error can occur when arguments to a function have side effects that impact other arguments to the function in the same call. Usually this kind of dependency is easy to avoid, but it can sometimes be obscured by dependencies that are difficult to discern, or by operator overloading. Consider this code example:

```C++
handle memory_handle;

memory_handle->acquire(*p);
```

This appears well-defined, but if `->` and `*` are overloaded operators, then this code is translated to something that resembles this:

```
```

```
C++

Handle::acquire(operator->(memory_handle), operator*(p));
```

And if there's a dependency between `operator->(memory_handle)` and `operator*(p)`, the code might rely on a specific evaluation order, even though the original code looks like there is no possible dependency.

volatile keyword default behavior

The MSVC compiler supports two different interpretations of the **`volatile`** storage qualifier that you can specify by using compiler switches. The /volatile:ms switch selects the Microsoft extended volatile semantics that guarantee strong ordering, as has been the traditional case for x86 and x64 because of the strong memory model on those architectures. The /volatile:iso switch selects the strict C++ standard volatile semantics that don't guarantee strong ordering.

On the ARM architecture (except ARM64EC), the default is **/volatile:iso** because ARM processors have a weakly ordered memory model, and because ARM software doesn't have a legacy of relying on the extended semantics of **/volatile:ms** and doesn't usually have to interface with software that does. However, it's still sometimes convenient or even required to compile an ARM program to use the extended semantics. For example, it may be too costly to port a program to use the ISO C++ semantics, or driver software might have to adhere to the traditional semantics to function correctly. In these cases, you can use the **/volatile:ms** switch; however, to recreate the traditional volatile semantics on ARM targets, the compiler must insert memory barriers around each read or write of a **`volatile`** variable to enforce strong ordering, which can have a negative impact on performance.

On the x86, x64 and ARM64EC architectures, the default is **/volatile:ms** because much of the software that has already been created for these architectures by using MSVC relies on them. When you compile x86, x64 and ARM64EC programs, you can specify the **/volatile:iso** switch to help avoid unnecessary reliance on the traditional volatile semantics, and to promote portability.

See also

Configure Visual C++ for ARM processors

Overview of ARM32 ABI Conventions

Article • 11/11/2021

The application binary interface (ABI) for code compiled for Windows on ARM processors is based on the standard ARM EABI. This article highlights key differences between Windows on ARM and the standard. This document covers the ARM32 ABI. For information about the ARM64 ABI, see Overview of ARM64 ABI conventions. For more information about the standard ARM EABI, see Application Binary Interface (ABI) for the ARM Architecture (external link).

Base Requirements

Windows on ARM always presumes that it's running on an ARMv7 architecture. Floating-point support in the form of VFPv3-D32 or later must be present in hardware. The VFP must support both single-precision and double-precision floating-point in hardware. The Windows runtime doesn't support emulation of floating-point to enable running on non-VFP hardware.

Advanced SIMD Extensions (NEON) support, including both integer and floating-point operations, must also be present in hardware. No run-time support for emulation is provided.

Integer divide support (UDIV/SDIV) is recommended but not required. Platforms that lack integer divide support may incur a performance penalty because these operations have to be trapped and possibly patched.

Endianness

Windows on ARM executes in little-endian mode. Both the MSVC compiler and the Windows runtime always expect little-endian data. The SETEND instruction in the ARM instruction set architecture (ISA) allows even user-mode code to change the current endianness. However, doing so is discouraged because it's dangerous for an application. If an exception is generated in big-endian mode, the behavior is unpredictable. It may lead to an application fault in user mode, or a bugcheck in kernel mode.

Alignment

Although Windows enables the ARM hardware to handle misaligned integer accesses transparently, alignment faults still may be generated in some situations. Follow these

rules for alignment:

- You don't have to align half-word-sized (16-bit) and word-sized (32-bit) integer loads and stores. The hardware handles them efficiently and transparently.
- Floating-point loads and stores should be aligned. The kernel handles unaligned loads and stores transparently, but with significant overhead.
- Load or store double (LDRD/STRD) and multiple (LDM/STM) operations should be aligned. The kernel handles most of them transparently, but with significant overhead.
- All uncached memory accesses must be aligned, even for integer accesses. Unaligned accesses cause an alignment fault.

Instruction Set

The instruction set for Windows on ARM is strictly limited to Thumb-2. All code executed on this platform is expected to start and always remain in Thumb mode. An attempt to switch into the legacy ARM instruction set may succeed. However, if it does, any exceptions or interrupts that occur may lead to an application fault in user mode, or a bugcheck in kernel mode.

A side-effect of this requirement is that all code pointers must have the low bit set. Then, when they're loaded and branched to via BLX or BX, the processor remains in Thumb mode. It doesn't try to execute the target code as 32-bit ARM instructions.

SDIV/UDIV instructions

The use of integer divide instructions SDIV and UDIV is fully supported, even on platforms without native hardware to handle them. The extra overhead per SDIV or UDIV divide on a Cortex-A9 processor is approximately 80 cycles. That's added to the overall divide time of 20-250 cycles, depending on the inputs.

Integer registers

The ARM processor supports 16 integer registers:

Register	Volatile?	Role
r0	Volatile	Parameter, result, scratch register 1
r1	Volatile	Parameter, result, scratch register 2

Register	Volatile?	Role
r2	Volatile	Parameter, scratch register 3
r3	Volatile	Parameter, scratch register 4
r4	Non-volatile	
r5	Non-volatile	
r6	Non-volatile	
r7	Non-volatile	
r8	Non-volatile	
r9	Non-volatile	
r10	Non-volatile	
r11	Non-volatile	Frame pointer
r12	Volatile	Intra-procedure-call scratch register
r13 (SP)	Non-volatile	Stack pointer
r14 (LR)	Non-volatile	Link register
r15 (PC)	Non-volatile	Program counter

For details about how to use the parameter and return value registers, see the Parameter Passing section in this article.

Windows uses r11 for fast-walking of the stack frame. For more information, see the Stack Walking section. Because of this requirement, r11 must always point to the topmost link in the chain. Don't use r11 for general purposes, because your code won't generate correct stack walks during analysis.

VFP registers

Windows only supports ARM variants that have VFPv3-D32 coprocessor support. It means floating-point registers are always present and can be relied on for parameter passing. And, the full set of 32 registers is available for use. The VFP registers and their usage are summarized in this table:

Singles	Doubles	Quads	Volatile?	Role
s0-s3	d0-d1	q0	Volatile	Parameters, result, scratch register

Singles	Doubles	Quads	Volatile?	Role
s4-s7	d2-d3	q1	Volatile	Parameters, scratch register
s8-s11	d4-d5	q2	Volatile	Parameters, scratch register
s12-s15	d6-d7	q3	Volatile	Parameters, scratch register
s16-s19	d8-d9	q4	Non-volatile	
s20-s23	d10-d11	q5	Non-volatile	
s24-s27	d12-d13	q6	Non-volatile	
s28-s31	d14-d15	q7	Non-volatile	
	d16-d31	q8-q15	Volatile	

The next table illustrates the floating-point status and control register (FPSCR) bitfields:

Bits	Meaning	Volatile?	Role
31-28	NZCV	Volatile	Status flags
27	QC	Volatile	Cumulative saturation
26	AHP	Non-volatile	Alternative half-precision control
25	DN	Non-volatile	Default NaN mode control
24	FZ	Non-volatile	Flush-to-zero mode control
23-22	RMode	Non-volatile	Rounding mode control
21-20	Stride	Non-volatile	Vector Stride, must always be 0
18-16	Len	Non-volatile	Vector Length, must always be 0
15, 12-8	IDE, IXE, and so on	Non-volatile	Exception trap enable bits, must always be 0
7, 4-0	IDC, IXC, and so on	Volatile	Cumulative exception flags

Floating-point exceptions

Most ARM hardware doesn't support IEEE floating-point exceptions. On processor variants that do have hardware floating-point exceptions, the Windows kernel silently catches the exceptions and implicitly disables them in the FPSCR register. This action ensures normalized behavior across processor variants. Otherwise, code developed on a

platform that doesn't have exception support could receive unexpected exceptions when it's running on a platform that does have exception support.

Parameter passing

The Windows on ARM ABI follows the ARM rules for parameter passing for non-variadic functions. The ABI rules include the VFP and Advanced SIMD extensions. These rules follow the Procedure Call Standard for the ARM Architecture, combined with the VFP extensions. By default, the first four integer arguments and up to eight floating-point or vector arguments are passed in registers. Any further arguments are passed on the stack. Arguments are assigned to registers or the stack by using this procedure:

Stage A: Initialization

Initialization is performed exactly once, before argument processing begins:

1. The Next Core Register Number (NCRN) is set to r0.
2. The VFP registers are marked as unallocated.
3. The Next Stacked Argument Address (NSAA) is set to the current SP.
4. If a function that returns a result in memory is called, then the address for the result is placed in r0 and the NCRN is set to r1.

Stage B: Pre-padding and extension of arguments

For each argument in the list, the first matching rule from the following list is applied:

1. If the argument is a composite type whose size cannot be statically determined by both the caller and the callee, the argument is copied to memory and replaced by a pointer to the copy.
2. If the argument is a byte or 16-bit half-word, then it is zero-extended or sign-extended to a 32-bit full word and treated as a 4-byte argument.
3. If the argument is a composite type, its size is rounded up to the nearest multiple of 4.

Stage C: Assignment of arguments to registers and stack

For each argument in the list, the following rules are applied in turn until the argument has been allocated:

1. If the argument is a VFP type and there are enough consecutive unallocated VFP registers of the appropriate type, then the argument is allocated to the lowest-numbered sequence of such registers.

2. If the argument is a VFP type, all remaining unallocated registers are marked as unavailable. The NSAA is adjusted upwards until it is correctly aligned for the argument type and the argument is copied to the stack at the adjusted NSAA. The NSAA is then incremented by the size of the argument.

3. If the argument requires 8-byte alignment, the NCRN is rounded up to the next even register number.

4. If the size of the argument in 32-bit words is not more than r4 minus NCRN, the argument is copied into core registers, starting at the NCRN, with the least significant bits occupying the lower-numbered registers. The NCRN is incremented by the number of registers used.

5. If the NCRN is less than r4 and the NSAA is equal to the SP, the argument is split between core registers and the stack. The first part of the argument is copied into the core registers, starting at the NCRN, up to and including r3. The rest of the argument is copied onto the stack, starting at the NSAA. The NCRN is set to r4 and the NSAA is incremented by the size of the argument minus the amount passed in registers.

6. If the argument requires 8-byte alignment, the NSAA is rounded up to the next 8-byte aligned address.

7. The argument is copied into memory at the NSAA. The NSAA is incremented by the size of the argument.

The VFP registers aren't used for variadic functions, and Stage C rules 1 and 2 are ignored. It means that a variadic function can begin with an optional push {r0-r3} to prepend the register arguments to any additional arguments passed by the caller, and then access the entire argument list directly from the stack.

Integer type values are returned in r0, optionally extended to r1 for 64-bit return values. VFP/NEON floating-point or SIMD type values are returned in s0, d0, or q0, as appropriate.

Stack

The stack must always remain 4-byte aligned, and must be 8-byte aligned at any function boundary. It's required to support the frequent use of interlocked operations

on 64-bit stack variables. The ARM EABI states that the stack is 8-byte aligned at any public interface. For consistency, the Windows on ARM ABI considers any function boundary to be a public interface.

Functions that have to use a frame pointer—for example, functions that call `alloca` or that change the stack pointer dynamically—must set up the frame pointer in r11 in the function prologue and leave it unchanged until the epilogue. Functions that don't require a frame pointer must perform all stack updates in the prologue and leave the stack pointer unchanged until the epilogue.

Functions that allocate 4 KB or more on the stack must ensure that each page prior to the final page is touched in order. This order ensures that no code can "leap over" the guard pages that Windows uses to expand the stack. Typically, the expansion is done by the `__chkstk` helper, which is passed the total stack allocation in bytes divided by 4 in r4, and which returns the final stack allocation amount in bytes back in r4.

Red zone

The 8-byte area immediately below the current stack pointer is reserved for analysis and dynamic patching. It permits carefully generated code to be inserted, which stores 2 registers at `[sp, #-8]` and temporarily uses them for arbitrary purposes. The Windows kernel guarantees that those 8 bytes won't be overwritten if an exception or interrupt occurs in both user mode and kernel mode.

Kernel stack

The default kernel-mode stack in Windows is three pages (12 KB). Be careful not to create functions that have large stack buffers in kernel mode. An interrupt could come in with very little stack headroom and cause a stack panic bugcheck.

C/C++ specifics

Enumerations are 32-bit integer types unless at least one value in the enumeration requires 64-bit double-word storage. In that case, the enumeration is promoted to a 64-bit integer type.

`wchar_t` is defined to be equivalent to **`unsigned short`**, to preserve compatibility with other platforms.

Stack walking

Windows code is compiled with frame pointers enabled (/Oy (Frame-Pointer Omission)) to enable fast stack walking. Generally, the r11 register points to the next link in the chain, which is an {r11, lr} pair that specifies the pointer to the previous frame on the stack and the return address. We recommend that your code also enable frame pointers for improved profiling and tracing.

Exception unwinding

Stack unwinding during exception handling is enabled by the use of unwind codes. The unwind codes are a sequence of bytes stored in the .xdata section of the executable image. They describe the operation of the function prologue and epilogue code in an abstract manner, so that the effects of a function's prologue can be undone in preparation for unwinding to the caller's stack frame.

The ARM EABI specifies an exception unwinding model that uses unwind codes. However, this specification isn't sufficient for unwinding in Windows, which must handle cases where the processor is in the middle of the prologue or epilogue of a function. For more information about Windows on ARM exception data and unwinding, see ARM Exception Handling.

We recommend that dynamically generated code be described by using dynamic function tables specified in calls to `RtlAddFunctionTable` and associated functions, so that the generated code can participate in exception handling.

Cycle counter

ARM processors running Windows are required to support a cycle counter, but using the counter directly may cause problems. To avoid these issues, Windows on ARM uses an undefined opcode to request a normalized 64-bit cycle-counter value. From C or C++, use the `__rdpmccntr64` intrinsic to emit the appropriate opcode; from assembly, use the `__rdpmccntr64` instruction. Reading the cycle counter takes approximately 60 cycles on a Cortex-A9.

The counter is a true cycle counter, not a clock; therefore, the counting frequency varies with the processor frequency. If you want to measure elapsed clock time, use `QueryPerformanceCounter`.

See also

Common Visual C++ ARM Migration Issues

ARM Exception Handling

Overview of ARM64 ABI conventions

Article • 08/25/2021

The basic application binary interface (ABI) for Windows when compiled and run on ARM processors in 64-bit mode (ARMv8 or later architectures), for the most part, follows ARM's standard AArch64 EABI. This article highlights some of the key assumptions and changes from what is documented in the EABI. For information about the 32-bit ABI, see Overview of ARM ABI conventions. For more information about the standard ARM EABI, see Application Binary Interface (ABI) for the ARM Architecture (external link).

Definitions

With the introduction of 64-bit support, ARM has defined several terms:

- **AArch32** – the legacy 32-bit instruction set architecture (ISA) defined by ARM, including Thumb mode execution.
- **AArch64** – the new 64-bit instruction set architecture (ISA) defined by ARM.
- **ARMv7** – the specification of the "7th generation" ARM hardware, which only includes support for AArch32. This version of the ARM hardware is the first version Windows for ARM supported.
- **ARMv8** – the specification of the "8th generation" ARM hardware, which includes support for both AArch32 and AArch64.

Windows also uses these terms:

- **ARM** – refers to the 32-bit ARM architecture (AArch32), sometimes referred to as WoA (Windows on ARM).
- **ARM32** – same as ARM, above; used in this document for clarity.
- **ARM64** – refers to the 64-bit ARM architecture (AArch64). There's no such thing as WoA64.

Finally, when referring to data types, the following definitions from ARM are referenced:

- **Short-Vector** – A data type directly representable in SIMD, a vector of 8 bytes or 16 bytes worth of elements. It's aligned to its size, either 8 bytes or 16 bytes, where each element can be 1, 2, 4, or 8 bytes.
- **HFA (Homogeneous Floating-point Aggregate)** – A data type with 2 to 4 identical floating-point members, either floats or doubles.
- **HVA (Homogeneous Short-Vector Aggregate)** – A data type with 2 to 4 identical Short-Vector members.

Base requirements

The ARM64 version of Windows presupposes that it's running on an ARMv8 or later architecture at all times. Both floating-point and NEON support are presumed to be present in hardware.

The ARMv8 specification describes new optional crypto and CRC helper opcodes for both AArch32 and AArch64. Support for them is currently optional, but recommended. To take advantage of these opcodes, apps should first make runtime checks for their existence.

Endianness

As with the ARM32 version of Windows, on ARM64 Windows executes in little-endian mode. Switching endianness is difficult to achieve without kernel mode support in AArch64, so it's easier to enforce.

Alignment

Windows running on ARM64 enables the CPU hardware to handle misaligned accesses transparently. In an improvement from AArch32, this support now also works for all integer accesses (including multi-word accesses) and for floating-point accesses.

However, accesses to uncached (device) memory still must always be aligned. If code could possibly read or write misaligned data from uncached memory, it must make sure to align all accesses.

Default layout alignment for locals:

Size in bytes	Alignment in bytes
1	1
2	2
3, 4	4
> 4	8

Default layout alignment for globals and statics:

Size in bytes	Alignment in bytes
1	1

Size in bytes	Alignment in bytes
2 - 7	4
8 - 63	8
>= 64	16

Integer registers

The AArch64 architecture supports 32 integer registers:

Register	Volatility	Role
x0-x8	Volatile	Parameter/Result scratch registers
x9-x15	Volatile	Scratch registers
x16-x17	Volatile	Intra-procedure-call scratch registers
x18	N/A	Reserved platform register: in kernel mode, points to KPCR for the current processor; In user mode, points to TEB
x19-x28	Non-volatile	Scratch registers
x29/fp	Non-volatile	Frame pointer
x30/lr	Both	Link Register: Callee function must preserve it for its own return, but caller's value will be lost.

Each register may be accessed as a full 64-bit value (via x0-x30) or as a 32-bit value (via w0-w30). 32-bit operations zero-extend their results up to 64 bits.

See the Parameter passing section for details on the use of the parameter registers.

Unlike AArch32, the program counter (PC) and the stack pointer (SP) aren't indexed registers. They're limited in how they may be accessed. Also note that there's no x31 register. That encoding is used for special purposes.

The frame pointer (x29) is required for compatibility with fast stack walking used by ETW and other services. It must point to the previous {x29, x30} pair on the stack.

Floating-point/SIMD registers

The AArch64 architecture also supports 32 floating-point/SIMD registers, summarized below:

Register	Volatility	Role
v0-v7	Volatile	Parameter/Result scratch registers
v8-v15	Both	Low 64 bits are Non-Volatile. High 64 bits are Volatile.
v16-v31	Volatile	Scratch registers

Each register may be accessed as a full 128-bit value (via v0-v31 or q0-q31). It may be accessed as a 64-bit value (via d0-d31), as a 32-bit value (via s0-s31), as a 16-bit value (via h0-h31), or as an 8-bit value (via b0-b31). Accesses smaller than 128 bits only access the lower bits of the full 128-bit register. They leave the remaining bits untouched unless otherwise specified. (AArch64 is different from AArch32, where the smaller registers were packed on top of the larger registers.)

The floating-point control register (FPCR) has certain requirements on the various bitfields within it:

Bits	Meaning	Volatility	Role
26	AHP	Non-Volatile	Alternative half-precision control.
25	DN	Non-Volatile	Default NaN mode control.
24	FZ	Non-volatile	Flush-to-zero mode control.
23-22	RMode	Non-volatile	Rounding mode control.
15,12-8	IDE/IXE/etc	Non-Volatile	Exception trap enable bits, must always be 0.

System registers

Like AArch32, the AArch64 specification provides three system-controlled "thread ID" registers:

Register	Role
TPIDR_EL0	Reserved.
TPIDRRO_EL0	Contains CPU number for current processor.
TPIDR_EL1	Points to KPCR structure for current processor.

Floating-point exceptions

Support for IEEE floating-point exceptions is optional on AArch64 systems. For processor variants that do have hardware floating-point exceptions, the Windows kernel silently catches the exceptions and implicitly disables them in the FPCR register. This trap ensures normalized behavior across processor variants. Otherwise, code developed on a platform without exception support may find itself taking unexpected exceptions when running on a platform with support.

Parameter passing

For non-variadic functions, the Windows ABI follows the rules specified by ARM for parameter passing. These rules are excerpted directly from the Procedure Call Standard for the AArch64 Architecture:

Stage A – Initialization

This stage is done exactly once, before processing of the arguments begins.

1. The Next General-purpose Register Number (NGRN) is set to zero.
2. The Next SIMD and Floating-point Register Number (NSRN) is set to zero.
3. The next stacked argument address (NSAA) is set to the current stack-pointer value (SP).

Stage B – Pre-padding and extension of arguments

For each argument in the list, the first matching rule from the following list is applied. If no rule matches, the argument is used unmodified.

1. If the argument type is a Composite Type whose size can't be statically determined by both the caller and the callee, the argument is copied to memory and the argument is replaced by a pointer to the copy. (There are no such types in C/C++ but they exist in other languages or in language extensions).
2. If the argument type is an HFA or an HVA, then the argument is used unmodified.
3. If the argument type is a Composite Type larger than 16 bytes, then the argument is copied to memory allocated by the caller, and the argument is replaced by a pointer to the copy.

4. If the argument type is a Composite Type, then the size of the argument is rounded up to the nearest multiple of 8 bytes.

Stage C – Assignment of arguments to registers and stack

For each argument in the list, the following rules are applied in turn until the argument has been allocated. When an argument is assigned to a register, any unused bits in the register have unspecified value. If an argument is assigned to a stack slot, any unused padding bytes have unspecified value.

1. If the argument is a Half-, Single-, Double- or Quad-precision Floating-point or Short Vector Type, and the NSRN is less than 8, then the argument is allocated to the least significant bits of register v[NSRN]. The NSRN is incremented by one. The argument has now been allocated.

2. If the argument is an HFA or an HVA, and there are sufficient unallocated SIMD and Floating-point registers (NSRN + number of members ≤ 8), then the argument is allocated to SIMD and Floating-point Registers, one register per member of the HFA or HVA. The NSRN is incremented by the number of registers used. The argument has now been allocated.

3. If the argument is an HFA or an HVA, then the NSRN is set to 8, and the size of the argument is rounded up to the nearest multiple of 8 bytes.

4. If the argument is an HFA, an HVA, a Quad-precision Floating-point or Short Vector Type, then the NSAA is rounded up to the larger of 8 or the Natural Alignment of the argument's type.

5. If the argument is a Half- or Single-precision Floating Point type, then the size of the argument is set to 8 bytes. The effect is as if the argument had been copied to the least significant bits of a 64-bit register, and the remaining bits filled with unspecified values.

6. If the argument is an HFA, an HVA, a Half-, Single-, Double-, or Quad-precision Floating-point or Short Vector Type, then the argument is copied to memory at the adjusted NSAA. The NSAA is incremented by the size of the argument. The argument has now been allocated.

7. If the argument is an Integral or Pointer Type, the size of the argument is less than or equal to 8 bytes, and the NGRN is less than 8, the argument is copied to the least significant bits in x[NGRN]. The NGRN is incremented by one. The argument has now been allocated.

8. If the argument has an alignment of 16, then the NGRN is rounded up to the next even number.

9. If the argument is an Integral Type, the size of the argument is equal to 16, and the NGRN is less than 7, the argument is copied to x[NGRN] and x[NGRN+1]. x[NGRN] shall contain the lower addressed double-word of the memory representation of the argument. The NGRN is incremented by two. The argument has now been allocated.

10. If the argument is a Composite Type, and the size in double-words of the argument is no more than 8 minus NGRN, then the argument is copied into consecutive general-purpose registers, starting at x[NGRN]. The argument is passed as though it had been loaded into the registers from a double-word-aligned address, with an appropriate sequence of LDR instructions that load consecutive registers from memory. The contents of any unused parts of the registers are unspecified by this standard. The NGRN is incremented by the number of registers used. The argument has now been allocated.

11. The NGRN is set to 8.

12. The NSAA is rounded up to the larger of 8 or the Natural Alignment of the argument's type.

13. If the argument is a composite type, then the argument is copied to memory at the adjusted NSAA. The NSAA is incremented by the size of the argument. The argument has now been allocated.

14. If the size of the argument is less than 8 bytes, then the size of the argument is set to 8 bytes. The effect is as if the argument was copied to the least significant bits of a 64-bit register, and the remaining bits were filled with unspecified values.

15. The argument is copied to memory at the adjusted NSAA. The NSAA is incremented by the size of the argument. The argument has now been allocated.

Addendum: Variadic functions

Functions that take a variable number of arguments are handled differently than above, as follows:

1. All composites are treated alike; no special treatment of HFAs or HVAs.

2. SIMD and Floating-point Registers aren't used.

Effectively, it's the same as following rules C.12–C.15 to allocate arguments to an imaginary stack, where the first 64 bytes of the stack are loaded into x0-x7, and any remaining stack arguments are placed normally.

Return values

Integral values are returned in x0.

Floating-point values are returned in s0, d0, or v0, as appropriate.

A type is considered to be an HFA or HVA if all of the following hold:

- It's non-empty,
- It doesn't have any non-trivial default or copy constructors, destructors, or assignment operators,
- All of its members have the same HFA or HVA type, or are float, double, or neon types that match the other members' HFA or HVA types.

HVA values with four or fewer elements are returned in s0-s3, d0-d3, or v0-v3, as appropriate.

Types returned by value are handled differently depending on whether they have certain properties, and whether the function is a non-static member function. Types which have all of these properties,

- they're *aggregate* by the C++14 standard definition, that is, they have no user-provided constructors, no private or protected non-static data members, no base classes, and no virtual functions, and
- they have a trivial copy-assignment operator, and
- they have a trivial destructor,

and are returned by non-member functions or static member functions, use the following return style:

- Types that are HFAs with four or fewer elements are returned in s0-s3, d0-d3, or v0-v3, as appropriate.
- Types less than or equal to 8 bytes are returned in x0.
- Types less than or equal to 16 bytes are returned in x0 and x1, with x0 containing the lower-order 8 bytes.
- For other aggregate types, the caller shall reserve a block of memory of sufficient size and alignment to hold the result. The address of the memory block shall be passed as an additional argument to the function in x8. The callee may modify the

result memory block at any point during the execution of the subroutine. The callee isn't required to preserve the value stored in x8.

All other types use this convention:

- The caller shall reserve a block of memory of sufficient size and alignment to hold the result. The address of the memory block shall be passed as an additional argument to the function in x0, or x1 if $this is passed in x0. The callee may modify the result memory block at any point during the execution of the subroutine. The callee returns the address of the memory block in x0.

Stack

Following the ABI put forth by ARM, the stack must remain 16-byte aligned at all times. AArch64 contains a hardware feature that generates stack alignment faults whenever the SP isn't 16-byte aligned and an SP-relative load or store is done. Windows runs with this feature enabled at all times.

Functions that allocate 4k or more worth of stack must ensure that each page prior to the final page is touched in order. This action ensures no code can "leap over" the guard pages that Windows uses to expand the stack. Typically the touching is done by the `__chkstk` helper, which has a custom calling convention that passes the total stack allocation divided by 16 in x15.

Red zone

The 16-byte area immediately below the current stack pointer is reserved for use by analysis and dynamic patching scenarios. This area permits carefully generated code to be inserted which stores two registers at [sp, #-16] and temporarily uses them for arbitrary purposes. The Windows kernel guarantees that those 16 bytes aren't overwritten if an exception or interrupt is taken, in both user and kernel mode.

Kernel stack

The default kernel mode stack in Windows is six pages (24k). Pay extra attention to functions with large stack buffers in kernel mode. An ill-timed interrupt could come in with little headroom and create a stack panic bug check.

Stack walking

Code within Windows is compiled with frame pointers enabled (/Oy-) to enable fast stack walking. Generally, x29 (fp) points to the next link in the chain, which is an {fp, lr} pair, indicating the pointer to the previous frame on the stack and the return address. Third-party code is encouraged to enable frame pointers as well, to allow for improved profiling and tracing.

Exception unwinding

Unwinding during exception handling is assisted through the use of unwind codes. The unwind codes are a sequence of bytes stored in the .xdata section of the executable. They describe the operation of the prologue and epilogue in an abstract manner, such that the effects of a function's prologue can be undone in preparation for backing up to the caller's stack frame. For more information on the unwind codes, see ARM64 exception handling.

The ARM EABI also specifies an exception unwinding model that uses unwind codes. However, the specification as presented is insufficient for unwinding in Windows, which must handle cases where the PC is in the middle of a function prologue or epilogue.

Code that is dynamically generated should be described with dynamic function tables via `RtlAddFunctionTable` and associated functions, so that the generated code can participate in exception handling.

Cycle counter

All ARMv8 CPUs are required to support a cycle counter register, a 64-bit register that Windows configures to be readable at any exception level, including user mode. It can be accessed via the special PMCCNTR_EL0 register, using the MSR opcode in assembly code, or the `_ReadStatusReg` intrinsic in C/C++ code.

The cycle counter here is a true cycle counter, not a wall clock. The counting frequency will vary with the processor frequency. If you feel you must know the frequency of the cycle counter, you shouldn't be using the cycle counter. Instead, you want to measure wall clock time, for which you should use `QueryPerformanceCounter`.

See also

Common Visual C++ ARM Migration Issues
ARM64 exception handling

Overview of ARM64EC ABI conventions

Article • 10/14/2022

ARM64EC is an application binary interface (ABI) that enables ARM64 binaries to run natively and interoperably with x64 code. Specifically, the ARM64EC ABI follows x64 software conventions including calling convention, stack usage, and data alignment, making ARM64EC and x64 code interoperable. The operating system emulates the x64 portion of the binary. (The EC in ARM64EC stands for *emulation compatible*.)

For more information on the x64 and ARM64 ABIs, see Overview of x64 ABI conventions and Overview of ARM64 ABI conventions.

ARM64EC doesn't solve memory model differences between x64 and ARM based architectures. For more information, see Common Visual C++ ARM migration issues.

Definitions

- **ARM64** - The code stream for ARM64 processes that contains traditional ARM64 code.
- **ARM64EC** - The code stream that utilizes a subset of the ARM64 register set to provide interoperability with x64 code.

Register mapping

x64 processes may have threads running ARM64EC code. So it's always possible to retrieve an x64 register context, ARM64EC uses a subset of the ARM64 core registers that map 1:1 to emulated x64 registers. Importantly, ARM64EC never uses registers outside of this subset, except to read the Thread Environment Block (TEB) address from `x18`.

Native ARM64 processes shouldn't regress in performance when some or many functions are recompiled as ARM64EC. To maintain performance, the ABI follows these principles:

- The ARM64EC register subset includes all registers that are part of the ARM64 function calling convention.
- The ARM64EC calling convention directly maps to the ARM64 calling convention.

Special helper routines like `__chkstk_arm64ec` use custom calling conventions and registers. These registers are also included in the ARM64EC subset of registers.

Register mapping for integer registers

ARM64EC register	x64 register	ARM64EC calling convention	ARM64 calling convention	x64 calling convention
`x0`	`rcx`	volatile	volatile	volatile

ARM64EC register	x64 register	ARM64EC calling convention	ARM64 calling convention	x64 calling convention
`x1`	`rdx`	volatile	volatile	volatile
`x2`	`r8`	volatile	volatile	volatile
`x3`	`r9`	volatile	volatile	volatile
`x4`	`r10`	volatile	volatile	volatile
`x5`	`r11`	volatile	volatile	volatile
`x6`	`mm1` (low 64 bits of x87 `R1` register)	volatile	volatile	volatile
`x7`	`mm2` (low 64 bits of x87 `R2` register)	volatile	volatile	volatile
`x8`	`rax`	volatile	volatile	volatile
`x9`	`mm3` (low 64 bits of x87 `R3` register)	volatile	volatile	volatile
`x10`	`mm4` (low 64 bits of x87 `R4` register)	volatile	volatile	volatile
`x11`	`mm5` (low 64 bits of x87 `R5` register)	volatile	volatile	volatile
`x12`	`mm6` (low 64 bits of x87 `R6` register)	volatile	volatile	volatile
`x13`	N/A	disallowed	volatile	N/A
`x14`	N/A	disallowed	volatile	N/A
`x15`	`mm7` (low 64 bits of x87 `R7` register)	volatile	volatile	volatile
`x16`	High 16 bits of each of the x87 `R0`-`R3` registers	volatile(`xip0`)	volatile(`xip0`)	volatile
`x17`	High 16 bits of each of the x87 `R4`-`R7` registers	volatile(`xip1`)	volatile(`xip1`)	volatile
`x18`	N/A	fixed(TEB)	fixed(TEB)	volatile
`x19`	`r12`	non-volatile	non-volatile	non-volatile
`x20`	`r13`	non-volatile	non-volatile	non-volatile
`x21`	`r14`	non-volatile	non-volatile	non-volatile
`x22`	`r15`	non-volatile	non-volatile	non-volatile
`x23`	N/A	disallowed	non-volatile	N/A

ARM64EC register	x64 register	ARM64EC calling convention	ARM64 calling convention	x64 calling convention
`x24`	N/A	disallowed	non-volatile	N/A
`x25`	`rsi`	non-volatile	non-volatile	non-volatile
`x26`	`rdi`	non-volatile	non-volatile	non-volatile
`x27`	`rbx`	non-volatile	non-volatile	non-volatile
`x28`	N/A	disallowed	disallowed	N/A
`fp`	`rbp`	non-volatile	non-volatile	non-volatile
`lr`	`mm0` (low 64 bits of x87 `R0` register)	volatile	volatile	N/A
`sp`	`rsp`	non-volatile	non-volatile	non-volatile
`pc`	`rip`	instruction pointer	instruction pointer	instruction pointer
`PSTATE` subset: `N`/`Z`/`C`/`V`/`SS` [1, 2]	`RFLAGS` subset: `SF`/`ZF`/`CF`/`OF`/`TF`	volatile	volatile	volatile
N/A	`RFLAGS` subset: `PF`/`AF`	N/A	N/A	volatile
N/A	`RFLAGS` subset: `DF`	N/A	N/A	non-volatile

[1] Avoid directly reading, writing or computing mappings between `PSTATE` and `RFLAGS`. These bits may be used in the future and are subject to change.

[2] The ARM64EC carry flag `C` is the inverse of the x64 carry flag `CF` for subtraction operations. There's no special handling, because the flag is volatile and is therefore trashed when transitioning between (ARM64EC and x64) functions.

Register mapping for vector registers

ARM64EC register	x64 register	ARM64EC calling convention	ARM64 calling convention	x64 calling convention
`v0`-`v5`	`xmm0`-`xmm5`	volatile	volatile	volatile
`v6`-`v7`	`xmm6`-`xmm7`	volatile	volatile	non-volatile
`v8`-`v15`	`xmm8`-`xmm15`	volatile [1]	volatile [1]	non-volatile
`v16`-`v31`	`xmm16`-`xmm31`	disallowed	volatile	disallowed (x64 emulator doesn't support AVX-512)
`FPCR` [2]	`MXCSR[15:6]`	non-volatile	non-volatile	non-volatile

ARM64EC register	x64 register	ARM64EC calling convention	ARM64 calling convention	x64 calling convention
`FPSR` [2]	`MXCSR[5:0]`	volatile	volatile	volatile

[1] These ARM64 registers are special in that the lower 64 bits are non-volatile but the upper 64 bits are volatile. From the point of view of an x64 caller, they're effectively volatile because the callee would trash data.

[2] Avoid directly reading, writing, or computing mappings of `FPCR` and `FPSR`. These bits may be used in the future and are subject to change.

Struct packing

ARM64EC follows the same struct packing rules used for x64 to ensure interoperability between ARM64EC code and x64 code. For more information and examples of x64 struct packing, see Overview of x64 ABI conventions.

Emulation helper ABI routines

ARM64EC code and thunks use emulation helper routines to transition between x64 and ARM64EC functions.

The following table describes each special ABI routine and the registers the ABI uses. The routines don't modify the listed preserved registers under the ABI column. No assumptions should be made about unlisted registers. On-disk, the ABI routine pointers are null. At load time, the loader updates the pointers to point to the x64 emulator routines.

Name	Description	ABI
`__os_arm64x_dispatch_call_no_redirect`	Called by an exit thunk to call an x64 target (either an x64 function or an x64 fast-forward sequence). The routine pushes the ARM64EC return address (in the `LR` register) followed by the address of the instruction that succeeds a `blr x16` instruction that invokes the x64 emulator. It then runs the `blr x16` instruction	return value in `x8` (`rax`)

Name	Description	ABI
`__os_arm64x_dispatch_ret`	Called by an entry thunk to return to its x64 caller. It pops the x64 return address from the stack and invokes the x64 emulator to jump to it	N/A
`__os_arm64x_check_call`	Called by ARM64EC code with a pointer to an exit thunk and the indirect ARM64EC target address to execute. The ARM64EC target is considered patchable, and execution always returns to the caller with either the same data it was called with, or with modified data	Arguments: `x9`: The target address `x10`: The exit thunk address `x11`: The fast forward sequence address Out: `x9`: If the target function was detoured, it contains the address of the fast forward sequence `x10`: The exit thunk address `x11`: If the function was detoured, it contains the exit thunk address. Otherwise, the target address jumped to Preserved registers: `x0`-`x8`, `x15` (`chkstk`). and `q0`-`q7`

Name	Description	ABI
`__os_arm64x_check_icall`	Called by ARM64EC code, with a pointer to an exit thunk, to handle a jump to either a target address that is either x64 or ARM64EC. If the target is x64 and the x64 code hasn't been patched, the routine sets the target address register. It points to the ARM64EC version of the function if one exists. Otherwise, it sets the register to point to the exit thunk that transitions to the x64 target. Then, it returns to the calling ARM64EC code, which then jumps to the address in the register. This routine is a non-optimized version of `__os_arm64x_check_call`, where the target address isn't known at compile time Used at a call-site of an indirect call	Arguments: `x9`: The target address `x10`: The exit thunk address `x11`: The fast forward sequence address Out: `x9`: If the target function was detoured, it contains the address of the fast forward sequence `x10`: The exit thunk address `x11`: If the function was detoured, it contains the exit thunk address. Otherwise, the target address jumped to Preserved registers: `x0`-`x8`, `x15` (`chkstk`), and `q0`-`q7`
`__os_arm64x_check_icall_cfg`	Same as `__os_arm64x_check_icall` but also checks that the specified address is a valid Control Flow Graph indirect call target	Arguments: `x10`: The address of the exit thunk `x11`: The address of the target function Out: `x9`: If the target is x64, the address to the function. Otherwise, undefined `x10`: The address of the exit thunk `x11`: If the target is x64, it contains the address of the exit thunk. Otherwise, the address of the function Preserved registers: `x0`-`x8`, `x15` (`chkstk`), and `q0`-`q7`
`__os_arm64x_get_x64_information`	Gets the requested part of the live x64 register context	`_Function_class_(ARM64X_GET_X64_INFORMATION) NTSTATUS LdrpGetX64Information(_In_ ULONG Type, _Out_ PVOID Output, _In_ PVOID ExtraInfo)`

Name	Description	ABI
`__os_arm64x_set_x64_information`	Sets the requested part of the live x64 register context	`_Function_class_(ARM64X_SET_X64_INFORMATION) NTSTATUS LdrpSetX64Information(_In_ ULONG Type,_In_ PVOID Input, _In_ PVOID ExtraInfo)`
`__os_arm64x_x64_jump`	Used in signature-less adjustor and other thunks that directly forward (`jmp`) a call to another function that can have any signature, deferring the potential application of the right thunk to the real target	Arguments: `x9`: target to jump to All parameter registers preserved (forwarded)

Thunks

Thunks are the low-level mechanisms to support ARM64EC and x64 functions calling each other. There are two types: *entry thunks* for entering ARM64EC functions and *exit thunks* for calling x64 functions.

Entry thunk and intrinsic entry thunks: x64 to ARM64EC function call

To support x64 callers when a C/C++ function is compiled as ARM64EC, the toolchain generates a single entry thunk consisting of ARM64EC machine code. Intrinsics have an entry thunk of their own. All other functions share an entry thunk with all functions that have a matching calling convention, parameters, and return type. The content of the thunk depends on the calling convention of the C/C++ function.

In addition to handling parameters and the return address, the thunk bridges the differences in volatility between ARM64EC and x64 vector registers caused by ARM64EC vector register mapping:

ARM64EC register	x64 register	ARM64EC calling convention	ARM64 calling convention	x64 calling convention
`v6`-`v15`	`xmm6`-`xmm15`	volatile, but saved/restored in the entry thunk (x64 to ARM64EC)	volatile or partially volatile upper 64 bits	non-volatile

The entry thunk performs the following actions:

Parameter number	Stack usage

Parameter number	Stack usage
0-4	Stores ARM64EC `v6` and `v7` into the caller-allocated home space Since the callee is ARM64EC, which doesn't have the notion of a home space, the stored values aren't clobbered. Allocates an extra 128 bytes on the stack and store ARM64EC `v8` through `v15`.
5-8	`x4` = 5th parameter from the stack `x5` = 6th parameter from the stack `x6` = 7th parameter from the stack `x7` = 8th parameter from the stack If the parameter is SIMD, the `v4`-`v7` registers are used instead
9+	Allocates `AlignUp(NumParams - 8 , 2) * 8` bytes on the stack. * Copies the 9th and remaining parameters to this area

* Aligning the value to an even number guarantees that the stack remains aligned to 16 bytes

If the function accepts a 32-bit integer parameter, the thunk is permitted to only push 32 bits instead of the full 64 bits of the parent register.

Next, the thunk uses an ARM64 `bl` instruction to call the ARM64EC function. After the function returns, the thunk:

1. Undoes any stack allocations
2. Calls the `__os_arm64x_dispatch_ret` emulator helper to pop the x64 return address and resume x64 emulation.

Exit thunk: ARM64EC to x64 function call

For every call that an ARM64EC C/C++ function makes to potential x64 code, the MSVC toolchain generates an exit thunk. The content of the thunk depends on the parameters of the x64 callee and whether the callee is using the standard calling convention or `__vectorcall`. The compiler obtains this information from a function declaration for the callee.

First, The thunk pushes the return address that's in the ARM64EC `lr` register and a dummy 8-byte value to guarantee that the stack is aligned to 16 bytes. Second, the thunk handles the parameters:

Parameter number	Stack usage
0-4	Allocates 32 bytes of home space on the stack

Parameter number	Stack usage
5-8	Allocates `AlignUp(NumParams - 4, 2) * 8` more bytes higher up on the stack. * Copies the 5th and any subsequent parameters from ARM64EC's `x4`-`x7` to this extra space
9+	Copies the 9th and remaining parameters to the extra space

* Aligning the value to an even number guarantees that the stack remains aligned to 16 bytes.

Third, the thunk calls the `__os_arm64x_dispatch_call_no_redirect` emulator helper to invoke the x64 emulator to run the x64 function. The call must be a `blr x16` instruction (conveniently, `x16` is a volatile register). A `blr x16` instruction is required because the x64 emulator parses this instruction as a hint.

The x64 function usually attempts to return to the emulator helper by using an x64 `ret` instruction. At this point, the x64 emulator detects that it's in ARM64EC code. It then reads the preceding 4-byte hint that happens to be the ARM64 `blr x16` instruction. Since this hint indicates that the return address is in this helper, the emulator jumps directly to this address.

The x64 function is permitted to return to the emulator helper using any branch instruction, including x64 `jmp` and `call`. The emulator handles these scenarios as well.

When the helper then returns to the thunk, the thunk:

1. Undoes any stack allocation
2. Pops the ARM64EC `lr` register
3. Executes an ARM64 `ret lr` instruction.

ARM64EC function name decoration

An ARM64EC function name has a secondary decoration applied after any language-specific decoration. For functions with C linkage (whether compiled as C or by using `extern "C"`), a `#` is prepended to the name. For C++ decorated functions, a `$$h` tag is inserted into the name.

```
foo          => #foo
?foo@@YAHXZ => ?foo@@$$hYAHXZ
```

`__vectorcall`

The ARM64EC toolchain currently doesn't support `__vectorcall`. The compiler emits an error when it detects `__vectorcall` usage with ARM64EC.

See also

Understanding ARM64EC ABI and assembly code
Common Visual C++ ARM migration issues
Decorated names

ARM Exception Handling

Article • 05/25/2022

Windows on ARM uses the same structured exception-handling mechanism for asynchronous hardware-generated exceptions and synchronous software-generated exceptions. Language-specific exception handlers are built on top of Windows structured exception handling by using language helper functions. This document describes exception handling in Windows on ARM, and the language helpers both the Microsoft ARM assembler and the MSVC compiler generate.

ARM Exception Handling

Windows on ARM uses *unwind codes* to control stack unwinding during structured exception handling (SEH). Unwind codes are a sequence of bytes stored in the `.xdata` section of the executable image. These codes describe the operation of function prologue and epilogue code in an abstract way. The handler uses them to undo the function prologue's effects when it unwinds to the caller's stack frame.

The ARM EABI (embedded application binary interface) specifies a model for exception unwinding that uses unwind codes. The model isn't sufficient for SEH unwinding in Windows. It must handle asynchronous cases where the processor is in the middle of the prologue or epilogue of a function. Windows also separates unwinding control into function-level unwinding and language-specific scope unwinding, which is unified in the ARM EABI. For these reasons, Windows on ARM specifies more details for the unwinding data and procedure.

Assumptions

Executable images for Windows on ARM use the Portable Executable (PE) format. For more information, see PE Format. Exception handling information is stored in the `.pdata` and `.xdata` sections of the image.

The exception handling mechanism makes certain assumptions about code that follows the ABI for Windows on ARM:

- When an exception occurs within the body of a function, the handler could undo the prologue's operations, or do the epilogue's operations in a forward manner. Both should produce identical results.

- Prologues and epilogues tend to mirror each other. This feature can be used to reduce the size of the metadata needed to describe unwinding.
- Functions tend to be relatively small. Several optimizations rely on this observation for efficient packing of data.
- If a condition is placed on an epilogue, it applies equally to each instruction in the epilogue.
- If the prologue saves the stack pointer (SP) in another register, that register must remain unchanged throughout the function, so the original SP may be recovered at any time.
- Unless the SP is saved in another register, all manipulation of it must occur strictly within the prologue and epilogue.
- To unwind any stack frame, these operations are required:
 - Adjust r13 (SP) in 4-byte increments.
 - Pop one or more integer registers.
 - Pop one or more VFP (virtual floating-point) registers.
 - Copy an arbitrary register value to r13 (SP).
 - Load SP from the stack by using a small post-decrement operation.
 - Parse one of a few well-defined frame types.

`.pdata` Records

The `.pdata` records in a PE-format image are an ordered array of fixed-length items that describe every stack-manipulating function. Leaf functions (functions that don't call other functions) don't require `.pdata` records when they don't manipulate the stack. (That is, they don't require any local storage and don't have to save or restore non-volatile registers.). Records for these functions can be omitted from the `.pdata` section to save space. An unwind operation from one of these functions can just copy the return address from the Link Register (LR) to the program counter (PC) to move up to the caller.

Every `.pdata` record for ARM is 8 bytes long. The general format of a record places the relative virtual address (RVA) of the function start in the first 32-bit word, followed by a second word that contains either a pointer to a variable-length `.xdata` block, or a

packed word that describes a canonical function unwinding sequence, as shown in this table:

Word Offset	Bits	Purpose
0	0-31	*`Function Start RVA`* is the 32-bit RVA of the start of the function. If the function contains thumb code, the low bit of this address must be set.
1	0-1	*`Flag`* is a 2-bit field that indicates how to interpret the remaining 30 bits of the second `.pdata` word. If *`Flag`* is 0, then the remaining bits form an *Exception Information RVA* (with the low two bits implicitly 0). If *`Flag`* is non-zero, then the remaining bits form a *Packed Unwind Data* structure.
1	2-31	*Exception Information RVA* or *Packed Unwind Data*. *Exception Information RVA* is the address of the variable-length exception information structure, stored in the `.xdata` section. This data must be 4-byte aligned. *Packed Unwind Data* is a compressed description of the operations required to unwind from a function, assuming a canonical form. In this case, no `.xdata` record is required.

Packed Unwind Data

For functions whose prologues and epilogues follow the canonical form described below, packed unwind data can be used. It eliminates the need for an `.xdata` record and significantly reduces the space required to provide the unwind data. The canonical prologues and epilogues are designed to meet the common requirements of a simple function that doesn't require an exception handler, and performs its setup and teardown operations in a standard order.

This table shows the format of a `.pdata` record that has packed unwind data:

Word Offset	Bits	Purpose
0	0-31	*`Function Start RVA`* is the 32-bit RVA of the start of the function. If the function contains thumb code, the low bit of this address must be set.

Word Offset	Bits	Purpose
1	0-1	*Flag* is a 2-bit field that has these meanings: - 00 = packed unwind data not used; remaining bits point to .xdata record. - 01 = packed unwind data. - 10 = packed unwind data where the function is assumed to have no prologue. This is useful for describing function fragments that are discontiguous with the start of the function. - 11 = reserved.
1	2-12	*Function Length* is an 11-bit field that provides the length of the entire function in bytes divided by 2. If the function is larger than 4K bytes, a full .xdata record must be used instead.
1	13-14	*Ret* is a 2-bit field that indicates how the function returns: - 00 = return via pop {pc} (the *L* flag bit must be set to 1 in this case). - 01 = return by using a 16-bit branch. - 10 = return by using a 32-bit branch. - 11 = no epilogue at all. This is useful for describing a discontiguous function fragment that may only contain a prologue, but whose epilogue is elsewhere.
1	15	*H* is a 1-bit flag that indicates whether the function "homes" the integer parameter registers (r0-r3) by pushing them at the start of the function, and deallocates the 16 bytes of stack before returning. (0 = doesn't home registers, 1 = homes registers.)
1	16-18	*Reg* is a 3-bit field that indicates the index of the last saved non-volatile register. If the *R* bit is 0, then only integer registers are being saved, and are assumed to be in the range of r4-rN, where N is equal to 4 + *Reg*. If the *R* bit is 1, then only floating-point registers are being saved, and are assumed to be in the range of d8-dN, where N is equal to 8 + *Reg*. The special combination of *R* = 1 and *Reg* = 7 indicates that no registers are saved.
1	19	*R* is a 1-bit flag that indicates whether the saved non-volatile registers are integer registers (0) or floating-point registers (1). If *R* is set to 1 and the *Reg* field is set to 7, no non-volatile registers were pushed.
1	20	*L* is a 1-bit flag that indicates whether the function saves/restores LR, along with other registers indicated by the *Reg* field. (0 = doesn't save/restore, 1 = does save/restore.)
1	21	*C* is a 1-bit flag that indicates whether the function includes extra instructions to set up a frame chain for fast stack walking (1) or not (0). If this bit is set, r11 is implicitly added to the list of integer non-volatile registers saved. (See restrictions below if the *C* flag is used.)

Word Offset	Bits	Purpose
1	22-31	`Stack Adjust` is a 10-bit field that indicates the number of bytes of stack that are allocated for this function, divided by 4. However, only values between 0x000-0x3F3 can be directly encoded. Functions that allocate more than 4044 bytes of stack must use a full `.xdata` record. If the `Stack Adjust` field is 0x3F4 or larger, then the low 4 bits have special meaning: - Bits 0-1 indicate the number of words of stack adjustment (1-4) minus 1. - Bit 2 is set to 1 if the prologue combined this adjustment into its push operation. - Bit 3 is set to 1 if the epilogue combined this adjustment into its pop operation.

Due to possible redundancies in the encodings above, these restrictions apply:

- If the `C` flag is set to 1:
 - The `L` flag must also be set to 1, because frame chaining requires both r11 and LR.
 - r11 must not be included in the set of registers described by `Reg`. That is, if r4-r11 are pushed, `Reg` should only describe r4-r10, because the `C` flag implies r11.
- If the `Ret` field is set to 0, the `L` flag must be set to 1.

Violating these restrictions causes an unsupported sequence.

For purposes of the discussion below, two pseudo-flags are derived from `Stack Adjust`:

- `PF` or "prologue folding" indicates that `Stack Adjust` is 0x3F4 or larger and bit 2 is set.
- `EF` or "epilogue folding" indicates that `Stack Adjust` is 0x3F4 or larger and bit 3 is set.

Prologues for canonical functions may have up to 5 instructions (notice that 3a and 3b are mutually exclusive):

Instruction	Opcode is assumed present if:	Size	Opcode	Unwind Codes
1	`H`==1	16	`push {r0-r3}`	04
2	`C`==1 or `L`==1 or `R`==0 or `PF`==1	16/32	`push {registers}`	80-BF/D0-DF/EC-ED
3a	`C`==1 and (`R`==1 and `PF`==0)	16	`mov r11,sp`	FB

Instruction	Opcode is assumed present if:	Size	Opcode	Unwind Codes
3b	`C`==1 and (`R`==0 or `PF`==1)	32	`add r11,sp,#xx`	FC
4	`R`==1 and `Reg` != 7	32	`vpush {d8-dE}`	E0-E7
5	`Stack Adjust` != 0 and `PF`==0	16/32	`sub sp,sp,#xx`	00-7F/E8-EB

Instruction 1 is always present if the `H` bit is set to 1.

To set up the frame chaining, either instruction 3a or 3b is present if the `C` bit is set. It is a 16-bit `mov` if no registers other than r11 and LR are pushed; otherwise, it is a 32-bit `add`.

If a non-folded adjustment is specified, instruction 5 is the explicit stack adjustment.

Instructions 2 and 4 are set based on whether a push is required. This table summarizes which registers are saved based on the `C`, `L`, `R`, and `PF` fields. In all cases, `N` is equal to `Reg` + 4, `E` is equal to `Reg` + 8, and `S` is equal to (~`Stack Adjust`) & 3.

C	L	R	PF	Integer Registers Pushed	VFP Registers pushed
0	0	0	0	r4 - r*`N`*	none
0	0	0	1	r*`S`* - r*`N`*	none
0	0	1	0	none	d8 - d*`E`*
0	0	1	1	r*`S`* - r3	d8 - d*`E`*
0	1	0	0	r4 - r*`N`*, LR	none
0	1	0	1	r*`S`* - r*`N`*, LR	none
0	1	1	0	LR	d8 - d*`E`*
0	1	1	1	r*`S`* - r3, LR	d8 - d*`E`*
1	0	0	0	(invalid encoding)	(invalid encoding)
1	0	0	1	(invalid encoding)	(invalid encoding)
1	0	1	0	(invalid encoding)	(invalid encoding)
1	0	1	1	(invalid encoding)	(invalid encoding)
1	1	0	0	r4 - r*`N`*, r11, LR	none
1	1	0	1	r*`S`* - r*`N`*, r11, LR	none

C	L	R	PF	Integer Registers Pushed	VFP Registers pushed
1	1	1	0	r11, LR	d8 - d*`E`*
1	1	1	1	r*`S`* - r3, r11, LR	d8 - d*`E`*

The epilogues for canonical functions follow a similar form, but in reverse and with some additional options. The epilogue may be up to 5 instructions long, and its form is strictly dictated by the form of the prologue.

Instruction	Opcode is assumed present if:	Size	Opcode
6	*`Stack Adjust`*!=0 and *`EF`*==0	16/32	`add sp,sp,#xx`
7	*`R`*==1 and *`Reg`*!=7	32	`vpop {d8-dE}`
8	*`C`*==1 or (*`L`*==1 and (*`H`*==0 or *`Ret`* !=0)) or *`R`*==0 or *`EF`*==1	16/32	`pop {registers}`
9a	*`H`*==1 and (*`L`*==0 or *`Ret`*!=0)	16	`add sp,sp,#0x10`
9b	*`H`*==1 and *`L`*==1 and *`Ret`*==0	32	`ldr pc, [sp],#0x14`
10a	*`Ret`*==1	16	`bx reg`
10b	*`Ret`*==2	32	`b address`

Instruction 6 is the explicit stack adjustment if a non-folded adjustment is specified. Because *`PF`* is independent of *`EF`*, it is possible to have instruction 5 present without instruction 6, or vice-versa.

Instructions 7 and 8 use the same logic as the prologue to determine which registers are restored from the stack, but with these three changes: first, *`EF`* is used in place of *`PF`*; second, if *`Ret`* = 0 and *`H`* = 0, then LR is replaced with PC in the register list and the epilogue ends immediately; third, if *`Ret`* = 0 and *`H`* = 1, then LR is omitted from the register list and popped by instruction 9b.

If *`H`* is set, then either instruction 9a or 9b is present. Instruction 9a is used when *`Ret`* is nonzero, which also implies the presence of either 10a or 10b. If L=1, then LR was popped as part of instruction 8. Instruction 9b is used when *`L`* is 1 and *`Ret`* is zero, to indicate an early end to the epilogue, and to return and adjust the stack at the same time.

If the epilogue hasn't already ended, then either instruction 10a or 10b is present, to indicate a 16-bit or 32-bit branch, based on the value of *`Ret`*.

`.xdata` Records

When the packed unwind format is insufficient to describe the unwinding of a function, a variable-length `.xdata` record must be created. The address of this record is stored in the second word of the `.pdata` record. The format of the `.xdata` is a packed variable-length set of words that has four sections:

1. A 1 or 2-word header that describes the overall size of the `.xdata` structure and provides key function data. The second word is only present if the *Epilogue Count* and *Code Words* fields are both set to 0. The fields are broken out in this table:

Word	Bits	Purpose
0	0-17	*`Function Length`* is an 18-bit field that indicates the total length of the function in bytes, divided by 2. If a function is larger than 512 KB, then multiple `.pdata` and `.xdata` records must be used to describe the function. For details, see the Large Functions section in this document.
0	18-19	*Vers* is a 2-bit field that describes the version of the remaining `.xdata`. Only version 0 is currently defined; values of 1-3 are reserved.
0	20	*X* is a 1-bit field that indicates the presence (1) or absence (0) of exception data.
0	21	*`E`* is a 1-bit field that indicates that information that describes a single epilogue is packed into the header (1) rather than requiring additional scope words later (0).
0	22	*F* is a 1-bit field that indicates that this record describes a function fragment (1) or a full function (0). A fragment implies that there's no prologue and that all prologue processing should be ignored.
0	23-27	*Epilogue Count* is a 5-bit field that has two meanings, depending on the state of the *`E`* bit: - If *`E`* is 0, this field is a count of the total number of epilogue scopes described in section 2. If more than 31 scopes exist in the function, then this field and the *Code Words* field must both be set to 0 to indicate that an extension word is required. - If *`E`* is 1, this field specifies the index of the first unwind code that describes the only epilogue.
0	28-31	*Code Words* is a 4-bit field that specifies the number of 32-bit words required to contain all of the unwind codes in section 4. If more than 15 words are required for more than 63 unwind code bytes, this field and the *Epilogue Count* field must both be set to 0 to indicate that an extension word is required.

Word	Bits	Purpose
1	0-15	*Extended Epilogue Count* is a 16-bit field that provides more space for encoding an unusually large number of epilogues. The extension word that contains this field is only present if the *Epilogue Count* and *Code Words* fields in the first header word are both set to 0.
1	16-23	*Extended Code Words* is an 8-bit field that provides more space for encoding an unusually large number of unwind code words. The extension word that contains this field is only present if the *Epilogue Count* and *Code Words* fields in the first header word are both set to 0.
1	24-31	Reserved

2. After the exception data (if the `E` bit in the header was set to 0) is a list of information about epilogue scopes, which are packed one to a word and stored in order of increasing starting offset. Each scope contains these fields:

Bits	Purpose
0-17	*Epilogue Start Offset* is an 18-bit field that describes the offset of the epilogue, in bytes divided by 2, relative to the start of the function.
18-19	*Res* is a 2-bit field reserved for future expansion. Its value must be 0.
20-23	*Condition* is a 4-bit field that gives the condition under which the epilogue is executed. For unconditional epilogues, it should be set to 0xE, which indicates "always". (An epilogue must be entirely conditional or entirely unconditional, and in Thumb-2 mode, the epilogue begins with the first instruction after the IT opcode.)
24-31	*Epilogue Start Index* is an 8-bit field that indicates the byte index of the first unwind code that describes this epilogue.

3. After the list of epilogue scopes comes an array of bytes that contain unwind codes, which are described in detail in the Unwind Codes section in this article. This array is padded at the end to the nearest full word boundary. The bytes are stored in little-endian order so that they can be directly fetched in little-endian mode.

4. If the *X* field in the header is 1, the unwind code bytes are followed by the exception handler information. This consists of one *Exception Handler RVA* that contains the address of the exception handler, followed immediately by the (variable-length) amount of data required by the exception handler.

The `.xdata` record is designed so that it is possible to fetch the first 8 bytes and compute the full size of the record, not including the length of the variable-sized

exception data that follows. This code snippet computes the record size:

C++

```
ULONG ComputeXdataSize(PULONG Xdata)
{
    ULONG Size;
    ULONG EpilogueScopes;
    ULONG UnwindWords;

    if ((Xdata[0] >> 23) != 0) {
        Size = 4;
        EpilogueScopes = (Xdata[0] >> 23) & 0x1f;
        UnwindWords = (Xdata[0] >> 28) & 0x0f;
    } else {
        Size = 8;
        EpilogueScopes = Xdata[1] & 0xffff;
        UnwindWords = (Xdata[1] >> 16) & 0xff;
    }

    if (!(Xdata[0] & (1 << 21))) {
        Size += 4 * EpilogueScopes;
    }

    Size += 4 * UnwindWords;

    if (Xdata[0] & (1 << 20)) {
        Size += 4;  // Exception handler RVA
    }

    return Size;
}
```

Although the prologue and each epilogue has an index into the unwind codes, the table is shared between them. It's not uncommon that they can all share the same unwind codes. We recommend that compiler writers optimize for this case, because the largest index that can be specified is 255, and that limits the total number of unwind codes possible for a particular function.

Unwind Codes

The array of unwind codes is a pool of instruction sequences that describe exactly how to undo the effects of the prologue, in the order in which the operations must be undone. The unwind codes are a mini instruction set, encoded as a string of bytes. When execution is complete, the return address to the calling function is in the LR register, and all non-volatile registers are restored to their values at the time the function was called.

If exceptions were guaranteed to only ever occur within a function body, and never within a prologue or epilogue, then only one unwind sequence would be necessary. However, the Windows unwinding model requires an ability to unwind from within a partially executed prologue or epilogue. To accommodate this requirement, the unwind codes have been carefully designed to have an unambiguous one-to-one mapping to each relevant opcode in the prologue and epilogue. This has several implications:

- It is possible to compute the length of the prologue and epilogue by counting the number of unwind codes. This is possible even with variable-length Thumb-2 instructions because there are distinct mappings for 16-bit and 32-bit opcodes.
- By counting the number of instructions past the start of an epilogue scope, it is possible to skip the equivalent number of unwind codes, and execute the rest of a sequence to complete the partially-executed unwind that the epilogue was performing.
- By counting the number of instructions before the end of the prologue, it is possible to skip the equivalent number of unwind codes, and execute the rest of the sequence to undo only those parts of the prologue that have completed execution.

The following table shows the mapping from unwind codes to opcodes. The most common codes are just one byte, while less common ones require two, three, or even four bytes. Each code is stored from most significant byte to least significant byte. The unwind code structure differs from the encoding described in the ARM EABI, because these unwind codes are designed to have a one-to-one mapping to the opcodes in the prologue and epilogue to allow for unwinding of partially executed prologues and epilogues.

Byte 1	Byte 2	Byte 3	Byte 4	Opsize	Explanation
00-7F				16	`add sp,sp,#X` where X is (Code & 0x7F) * 4
80-BF	00-FF			32	`pop {r0-r12, lr}` where LR is popped if Code & 0x2000 and r0-r12 are popped if the corresponding bit is set in Code & 0x1FFF
C0-CF				16	`mov sp,rX` where X is Code & 0x0F

Byte 1	Byte 2	Byte 3	Byte 4	Opsize	Explanation
D0-D7				16	`pop {r4-rX,lr}` where X is (Code & 0x03) + 4 and LR is popped if Code & 0x04
D8-DF				32	`pop {r4-rX,lr}` where X is (Code & 0x03) + 8 and LR is popped if Code & 0x04
E0-E7				32	`vpop {d8-dX}` where X is (Code & 0x07) + 8
E8-EB	00-FF			32	`addw sp,sp,#X` where X is (Code & 0x03FF) * 4
EC-ED	00-FF			16	`pop {r0-r7,lr}` where LR is popped if Code & 0x0100 and r0-r7 are popped if the corresponding bit is set in Code & 0x00FF
EE	00-0F			16	Microsoft-specific
EE	10-FF			16	Available
EF	00-0F			32	`ldr lr,[sp],#X` where X is (Code & 0x000F) * 4
EF	10-FF			32	Available
F0-F4				-	Available
F5	00-FF			32	`vpop {dS-dE}` where S is (Code & 0x00F0) >> 4 and E is Code & 0x000F
F6	00-FF			32	`vpop {dS-dE}` where S is ((Code & 0x00F0) >> 4) + 16 and E is (Code & 0x000F) + 16

Byte 1	Byte 2	Byte 3	Byte 4	Opsize	Explanation
F7	00-FF	00-FF		16	`add sp,sp,#X` where X is (Code & 0x00FFFF) * 4
F8	00-FF	00-FF	00-FF	16	`add sp,sp,#X` where X is (Code & 0x00FFFFFF) * 4
F9	00-FF	00-FF		32	`add sp,sp,#X` where X is (Code & 0x00FFFF) * 4
FA	00-FF	00-FF	00-FF	32	`add sp,sp,#X` where X is (Code & 0x00FFFFFF) * 4
FB				16	nop (16-bit)
FC				32	nop (32-bit)
FD				16	end + 16-bit nop in epilogue
FE				32	end + 32-bit nop in epilogue
FF				-	end

This shows the range of hexadecimal values for each byte in an unwind code *Code*, along with the opcode size *Opsize* and the corresponding original instruction interpretation. Empty cells indicate shorter unwind codes. In instructions that have large values covering multiple bytes, the most significant bits are stored first. The *Opsize* field shows the implicit opcode size associated with each Thumb-2 operation. The apparent duplicate entries in the table with different encodings are used to distinguish between different opcode sizes.

The unwind codes are designed so that the first byte of the code tells both the total size in bytes of the code and the size of the corresponding opcode in the instruction stream. To compute the size of the prologue or epilogue, walk the unwind codes from the start of the sequence to the end, and use a lookup table or similar method to determine how long the corresponding opcode is.

Unwind codes 0xFD and 0xFE are equivalent to the regular end code 0xFF, but account for one extra nop opcode in the epilogue case, either 16-bit or 32-bit. For prologues, codes 0xFD, 0xFE and 0xFF are exactly equivalent. This accounts for the common epilogue endings `bx lr` or `b <tailcall-target>`, which don't have an equivalent

prologue instruction. This increases the chance that unwind sequences can be shared between the prologue and the epilogues.

In many cases, it should be possible to use the same set of unwind codes for the prologue and all epilogues. However, to handle the unwinding of partially executed prologues and epilogues, you might have to have multiple unwind code sequences that vary in ordering or behavior. This is why each epilogue has its own index into the unwind array to show where to begin executing.

Unwinding Partial Prologues and Epilogues

The most common unwinding case is when the exception occurs in the body of the function, away from the prologue and all epilogues. In this case, the unwinder executes the codes in the unwind array beginning at index 0 and continues until an end opcode is detected.

When an exception occurs while a prologue or epilogue is executing, the stack frame is only partially constructed, and the unwinder must determine exactly what has been done in order to correctly undo it.

For example, consider this prologue and epilogue sequence:

asm

```
0000:   push  {r0-r3}         ; 0x04
0002:   push  {r4-r9, lr}     ; 0xdd
0006:   mov   r7, sp          ; 0xc7
...
0140:   mov   sp, r7          ; 0xc7
0142:   pop   {r4-r9, lr}     ; 0xdd
0146:   add   sp, sp, #16     ; 0x04
0148:   bx    lr
```

Next to each opcode is the appropriate unwind code to describe this operation. The sequence of unwind codes for the prologue is a mirror image of the unwind codes for the epilogue, not counting the final instruction. This case is common, and is the reason the unwind codes for the prologue are always assumed to be stored in reverse order from the prologue's execution order. This gives us a common set of unwind codes:

asm

```
0xc7, 0xdd, 0x04, 0xfd
```

The 0xFD code is a special code for the end of the sequence that means that the epilogue is one 16-bit instruction longer than the prologue. This makes greater sharing of unwind codes possible.

In the example, if an exception occurs while the function body between the prologue and epilogue is executing, unwinding starts with the epilogue case, at offset 0 within the epilogue code. This corresponds to offset 0x140 in the example. The unwinder executes the full unwind sequence, because no cleanup has been done. If instead the exception occurs one instruction after the beginning of the epilogue code, the unwinder can successfully unwind by skipping the first unwind code. Given a one-to-one mapping between opcodes and unwind codes, if unwinding from instruction *n* in the epilogue, the unwinder should skip the first *n* unwind codes.

Similar logic works in reverse for the prologue. If unwinding from offset 0 in the prologue, nothing has to be executed. If unwinding from one instruction in, the unwind sequence should start one unwind code from the end because prologue unwind codes are stored in reverse order. In the general case, if unwinding from instruction *n* in the prologue, unwinding should start executing at *n* unwind codes from the end of the list of codes.

Prologue and epilogue unwind codes don't always match exactly. In that case, the unwind code array may have to contain several sequences of codes. To determine the offset to begin processing codes, use this logic:

1. If unwinding from within the body of the function, begin executing unwind codes at index 0 and continue until an end opcode is reached.

2. If unwinding from within an epilogue, use the epilogue-specific starting index provided by the epilogue scope. Calculate how many bytes the PC is from the start of the epilogue. Skip forward through the unwind codes until all of the already-executed instructions are accounted for. Execute the unwind sequence starting at that point.

3. If unwinding from within the prologue, start from index 0 in the unwind codes. Calculate the length of the prologue code from the sequence, and then calculate how many bytes the PC is from the end of the prologue. Skip forward through the unwind codes until all of the unexecuted instructions are accounted for. Execute the unwind sequence starting at that point.

The unwind codes for the prologue must always be the first in the array. they're also the codes used to unwind in the general case of unwinding from within the body. Any epilogue-specific code sequences should follow immediately after the prologue code sequence.

Function Fragments

For code optimization, it may be useful to split a function into discontiguous parts. When this is done, each function fragment requires its own separate `.pdata`—and possibly `.xdata`—record.

Assuming that the function prologue is at the beginning of the function and can't be split, there are four function fragment cases:

- Prologue only; all epilogues in other fragments.
- Prologue and one or more epilogues; more epilogues in other fragments.
- No prologue or epilogues; prologue and one or more epilogues in other fragments.
- Epilogues only; prologue and possibly more epilogues in other fragments.

In the first case, only the prologue must be described. This can be done in compact `.pdata` form by describing the prologue normally and specifying a *`Ret`* value of 3 to indicate no epilogue. In the full `.xdata` form, this can be done by providing the prologue unwind codes at index 0 as usual, and specifying an epilogue count of 0.

The second case is just like a normal function. If there's only one epilogue in the fragment, and it is at the end of the fragment, then a compact `.pdata` record can be used. Otherwise, a full `.xdata` record must be used. Keep in mind that the offsets specified for the epilogue start are relative to the start of the fragment, not the original start of the function.

The third and fourth cases are variants of the first and second cases, respectively, except they don't contain a prologue. In these situations, it is assumed that there's code before the start of the epilogue and it is considered part of the body of the function, which would normally be unwound by undoing the effects of the prologue. These cases must therefore be encoded with a pseudo-prologue, which describes how to unwind from within the body, but which is treated as 0-length when determining whether to perform a partial unwind at the start of the fragment. Alternatively, this pseudo-prologue may be described by using the same unwind codes as the epilogue because they presumably perform equivalent operations.

In the third and fourth cases, the presence of a pseudo-prologue is specified either by setting the *`Flag`* field of the compact `.pdata` record to 2, or by setting the *F* flag in the `.xdata` header to 1. In either case, the check for a partial prologue unwind is ignored, and all non-epilogue unwinds are considered to be full.

Large Functions

Fragments can be used to describe functions larger than the 512 KB limit imposed by the bit fields in the `.xdata` header. To describe a larger function, just break it into fragments smaller than 512 KB. Each fragment should be adjusted so it doesn't split an epilogue into multiple pieces.

Only the first fragment of the function contains a prologue. All other fragments are marked as having no prologue. Depending on the number of epilogues, each fragment may contain zero or more epilogues. Keep in mind that each epilogue scope in a fragment specifies its starting offset relative to the start of the fragment, not the start of the function.

If a fragment has no prologue and no epilogue, it still requires its own `.pdata`—and possibly `.xdata`—record to describe how to unwind from within the body of the function.

Shrink-wrapping

A more complex special case of function fragments is called *shrink-wrapping*. It's a technique for deferring register saves from the start of the function to later in the function. It optimizes for simple cases that don't require register saving. This case has two parts: there's an outer region that allocates the stack space but saves a minimal set of registers, and an inner region that saves and restores other registers.

asm

```
ShrinkWrappedFunction
    push   {r4, lr}          ; A: save minimal non-volatiles
    sub    sp, sp, #0x100    ; A: allocate all stack space up front
    ...                      ; A:
    add    r0, sp, #0xE4     ; A: prepare to do the inner save
    stm    r0, {r5-r11}      ; A: save remaining non-volatiles
    ...                      ; B:
    add    r0, sp, #0xE4     ; B: prepare to do the inner restore
    ldm    r0, {r5-r11}      ; B: restore remaining non-volatiles
    ...                      ; C:
    pop    {r4, pc}          ; C:
```

Shrink-wrapped functions are typically expected to pre-allocate the space for the extra register saves in the regular prologue, and then save the registers by using `str` or `stm` instead of `push`. This action keeps all stack-pointer manipulation in the function's original prologue.

The example shrink-wrapped function must be broken into three regions, which are marked as `A`, `B`, and `C` in the comments. The first `A` region covers the start of the function through the end of the additional non-volatile saves. A `.pdata` or `.xdata` record must be constructed to describe this fragment as having a prologue and no epilogues.

The middle `B` region gets its own `.pdata` or `.xdata` record that describes a fragment that has no prologue and no epilogue. However, the unwind codes for this region must still be present because it's considered a function body. The codes must describe a composite prologue that represents both the original registers saved in the region `A` prologue and the extra registers saved before entering region `B`, as if they were produced by one sequence of operations.

The register saves for region `B` can't be considered as an "inner prologue" because the composite prologue described for region `B` must describe both the region `A` prologue and the additional registers saved. If fragment `B` had a prologue, the unwind codes would also imply the size of that prologue, and there's no way to describe the composite prologue in a way that maps one-to-one with the opcodes that only save the additional registers.

The extra register saves must be considered part of region `A`, because until they're complete, the composite prologue doesn't accurately describe the state of the stack.

The last `C` region gets its own `.pdata` or `.xdata` record, describing a fragment that has no prologue but does have an epilogue.

An alternative approach can also work if the stack manipulation done before entering region `B` can be reduced to one instruction:

asm

```
ShrinkWrappedFunction
    push   {r4, lr}          ; A: save minimal non-volatile registers
    sub    sp, sp, #0xE0     ; A: allocate minimal stack space up front
    ...                      ; A:
    push   {r4-r9}           ; A: save remaining non-volatiles
    ...                      ; B:
    pop    {r4-r9}           ; B: restore remaining non-volatiles
    ...                      ; C:
    pop    {r4, pc}          ; C: restore non-volatile registers
```

The key insight is that on each instruction boundary, the stack is fully consistent with the unwind codes for the region. If an unwind occurs before the inner push in this example, it's considered part of region `A`. Only the region `A` prologue is unwound. If the unwind

occurs after the inner push, it's considered part of region `B`, which has no prologue. However, it has unwind codes that describe both the inner push and the original prologue from region `A`. Similar logic holds for the inner pop.

Encoding Optimizations

The richness of the unwind codes, and the ability to make use of compact and expanded forms of data, provide many opportunities to optimize the encoding to further reduce space. With aggressive use of these techniques, the net overhead of describing functions and fragments by using unwind codes can be minimized.

The most important optimization idea: Don't confuse prologue and epilogue boundaries for unwinding purposes with logical prologue and epilogue boundaries from a compiler perspective. The unwinding boundaries can be shrunk and made tighter to improve efficiency. For example, a prologue may contain code after the stack setup to do verification checks. But once all the stack manipulation is complete, there's no need to encode further operations, and anything beyond that can be removed from the unwinding prologue.

This same rule applies to the function length. If there's data (such as a literal pool) that follows an epilogue in a function, it shouldn't be included as part of the function length. By shrinking the function to just the code that's part of the function, the chances are much greater that the epilogue is at the very end and a compact `.pdata` record can be used.

In a prologue, once the stack pointer is saved to another register, there's typically no need to record any further opcodes. To unwind the function, the first thing that's done is to recover SP from the saved register. Further operations don't have any effect on the unwind.

Single-instruction epilogues don't have to be encoded at all, either as scopes or as unwind codes. If an unwind takes place before that instruction is executed, then it's safe to assume it's from within the body of the function. Just executing the prologue unwind codes is sufficient. When the unwind takes place after the single instruction is executed, then by definition it takes place in another region.

Multi-instruction epilogues don't have to encode the first instruction of the epilogue, for the same reason as the previous point: if the unwind takes place before that instruction executes, a full prologue unwind is sufficient. If the unwind takes place after that instruction, then only the later operations have to be considered.

Unwind code reuse should be aggressive. The index each epilogue scope specifies points to an arbitrary starting point in the array of unwind codes. It doesn't have to point to the start of a previous sequence; it can point in the middle. The best approach is to generate the unwind code sequence. Then, scan for an exact byte match in the already-encoded pool of sequences. Use any perfect match as a starting point for reuse.

After single-instruction epilogues are ignored, if there are no remaining epilogues, consider using a compact `.pdata` form; it becomes much more likely in the absence of an epilogue.

Examples

In these examples, the image base is at 0x00400000.

Example 1: Leaf Function, No Locals

asm

```
Prologue:
  004535F8: B430      push        {r4-r5}
Epilogue:
  00453656: BC30      pop         {r4-r5}
  00453658: 4770      bx          lr
```

`.pdata` (fixed, 2 words):

- Word 0
 - *`Function Start RVA`* = 0x000535F8 (= 0x004535F8-0x00400000)
- Word 1
 - *`Flag`* = 1, indicating canonical prologue and epilogue formats
 - *`Function Length`* = 0x31 (= 0x62/2)
 - *`Ret`* = 1, indicating a 16-bit branch return
 - *`H`* = 0, indicating the parameters weren't homed
 - *`R`* = 0 and *`Reg`* = 1, indicating push/pop of r4-r5
 - *`L`* = 0, indicating no LR save/restore
 - *`C`* = 0, indicating no frame chaining

- `Stack Adjust` = 0, indicating no stack adjustment

Example 2: Nested Function with Local Allocation

```asm
Prologue:
  004533AC: B5F0      push        {r4-r7, lr}
  004533AE: B083      sub         sp, sp, #0xC
Epilogue:
  00453412: B003      add         sp, sp, #0xC
  00453414: BDF0      pop         {r4-r7, pc}
```

`.pdata` (fixed, 2 words):

- Word 0
 - `Function Start RVA` = 0x000533AC (= 0x004533AC -0x00400000)
- Word 1
 - `Flag` = 1, indicating canonical prologue and epilogue formats
 - `Function Length` = 0x35 (= 0x6A/2)
 - `Ret` = 0, indicating a pop {pc} return
 - `H` = 0, indicating the parameters weren't homed
 - `R` = 0 and `Reg` = 3, indicating push/pop of r4-r7
 - `L` = 1, indicating LR was saved/restored
 - `C` = 0, indicating no frame chaining
 - `Stack Adjust` = 3 (= 0x0C/4)

Example 3: Nested Variadic Function

```asm
Prologue:
  00453988: B40F      push        {r0-r3}
  0045398A: B570      push        {r4-r6, lr}
Epilogue:
  004539D4: E8BD 4070 pop         {r4-r6}
  004539D8: F85D FB14 ldr         pc, [sp], #0x14
```

`.pdata` (fixed, 2 words):

- Word 0
 - *`Function Start RVA`* = 0x00053988 (= 0x00453988-0x00400000)
- Word 1
 - *`Flag`* = 1, indicating canonical prologue and epilogue formats
 - *`Function Length`* = 0x2A (= 0x54/2)
 - *`Ret`* = 0, indicating a pop {pc}-style return (in this case an `ldr pc,[sp],#0x14` return)
 - *`H`* = 1, indicating the parameters were homed
 - *`R`* = 0 and *`Reg`* = 2, indicating push/pop of r4-r6
 - *`L`* = 1, indicating LR was saved/restored
 - *`C`* = 0, indicating no frame chaining
 - *`Stack Adjust`* = 0, indicating no stack adjustment

Example 4: Function with Multiple Epilogues

asm

```
Prologue:
  004592F4: E92D 47F0 stmdb       sp!, {r4-r10, lr}
  004592F8: B086      sub         sp, sp, #0x18
Epilogues:
  00459316: B006      add         sp, sp, #0x18
  00459318: E8BD 87F0 ldm         sp!, {r4-r10, pc}
  ...
  0045943E: B006      add         sp, sp, #0x18
  00459440: E8BD 87F0 ldm         sp!, {r4-r10, pc}
  ...
  004595D4: B006      add         sp, sp, #0x18
  004595D6: E8BD 87F0 ldm         sp!, {r4-r10, pc}
  ...
  00459606: B006      add         sp, sp, #0x18
  00459608: E8BD 87F0 ldm         sp!, {r4-r10, pc}
  ...
  00459636: F028 FF0F bl          KeBugCheckEx     ; end of function
```

`.pdata` (fixed, 2 words):

- Word 0
 - `Function Start RVA` = 0x000592F4 (= 0x004592F4-0x00400000)
- Word 1
 - `Flag` = 0, indicating `.xdata` record present (required for multiple epilogues)
 - `.xdata` *address* - 0x00400000

`.xdata` (variable, 6 words):

- Word 0
 - `Function Length` = 0x0001A3 (= 0x000346/2)
 - `Vers` = 0, indicating the first version of `.xdata`
 - `X` = 0, indicating no exception data
 - `E` = 0, indicating a list of epilogue scopes
 - `F` = 0, indicating a full function description, including prologue
 - `Epilogue Count` = 0x04, indicating the 4 total epilogue scopes
 - `Code Words` = 0x01, indicating one 32-bit word of unwind codes
- Words 1-4, describing 4 epilogue scopes at 4 locations. Each scope has a common set of unwind codes, shared with the prologue, at offset 0x00, and is unconditional, specifying condition 0x0E (always).
- Unwind codes, starting at Word 5: (shared between prologue/epilogue)
 - Unwind code 0 = 0x06: sp += (6 << 2)
 - Unwind code 1 = 0xDE: pop {r4-r10, lr}
 - Unwind code 2 = 0xFF: end

Example 5: Function with Dynamic Stack and Inner Epilogue

asm

```
Prologue:
  00485A20: B40F      push        {r0-r3}
  00485A22: E92D 41F0 stmdb       sp!, {r4-r8, lr}
```

```
  00485A26: 466E      mov         r6, sp
  00485A28: 0934      lsrs        r4, r6, #4
  00485A2A: 0124      lsls        r4, r4, #4
  00485A2C: 46A5      mov         sp, r4
  00485A2E: F2AD 2D90 subw        sp, sp, #0x290
Epilogue:
  00485BAC: 46B5      mov         sp, r6
  00485BAE: E8BD 41F0 ldm         sp!, {r4-r8, lr}
  00485BB2: B004      add         sp, sp, #0x10
  00485BB4: 4770      bx          lr
  ...
  00485E2A: F7FF BE7D b           #0x485B28    ; end of function
```

`.pdata` (fixed, 2 words):

- Word 0
 - *`Function Start RVA`* = 0x00085A20 (= 0x00485A20-0x00400000)
- Word 1
 - *`Flag`* = 0, indicating `.xdata` record present (needed for multiple epilogues)
 - *`.xdata`* *address* - 0x00400000

`.xdata` (variable, 3 words):

- Word 0
 - *`Function Length`* = 0x0001A3 (= 0x000346/2)
 - *`Vers`* = 0, indicating the first version of `.xdata`
 - *`X`* = 0, indicating no exception data
 - *`E`* = 0, indicating a list of epilogue scopes
 - *`F`* = 0, indicating a full function description, including prologue
 - *`Epilogue Count`* = 0x001, indicating the 1 total epilogue scope
 - *`Code Words`* = 0x01, indicating one 32-bit word of unwind codes
- Word 1: Epilogue scope at offset 0xC6 (= 0x18C/2), starting unwind code index at 0x00, and with a condition of 0x0E (always)
- Unwind codes, starting at Word 2: (shared between prologue/epilogue)
 - Unwind code 0 = 0xC6: sp = r6

- Unwind code 1 = 0xDC: pop {r4-r8, lr}
- Unwind code 2 = 0x04: sp += (4 << 2)
- Unwind code 3 = 0xFD: end, counts as 16-bit instruction for epilogue

Example 6: Function with Exception Handler

asm

```
Prologue:
  00488C1C: 0059 A7ED dc.w  0x0059A7ED
  00488C20: 005A 8ED0 dc.w  0x005A8ED0
FunctionStart:
  00488C24: B590      push        {r4, r7, lr}
  00488C26: B085      sub         sp, sp, #0x14
  00488C28: 466F      mov         r7, sp
Epilogue:
  00488C6C: 46BD      mov         sp, r7
  00488C6E: B005      add         sp, sp, #0x14
  00488C70: BD90      pop         {r4, r7, pc}
```

`.pdata` (fixed, 2 words):

- Word 0
 - *`Function Start RVA`* = 0x00088C24 (= 0x00488C24-0x00400000)
- Word 1
 - *`Flag`* = 0, indicating `.xdata` record present (needed for multiple epilogues)
 - `.xdata` *address* - 0x00400000

`.xdata` (variable, 5 words):

- Word 0
 - *`Function Length`* =0x000027 (= 0x00004E/2)
 - *`Vers`* = 0, indicating the first version of `.xdata`
 - *`X`* = 1, indicating exception data present
 - *`E`* = 1, indicating a single epilogue
 - *`F`* = 0, indicating a full function description, including prologue

 - `Epilogue Count` = 0x00, indicating epilogue unwind codes start at offset 0x00
 - `Code Words` = 0x02, indicating two 32-bit words of unwind codes
- Unwind codes, starting at Word 1:
 - Unwind code 0 = 0xC7: sp = r7
 - Unwind code 1 = 0x05: sp += (5 << 2)
 - Unwind code 2 = 0xED/0x90: pop {r4, r7, lr}
 - Unwind code 4 = 0xFF: end
- Word 3 specifies an exception handler = 0x0019A7ED (= 0x0059A7ED - 0x00400000)
- Words 4 and beyond are inlined exception data

Example 7: Funclet

asm

```
Function:
  00488C72: B500      push        {lr}
  00488C74: B081      sub         sp, sp, #4
  00488C76: 3F20      subs        r7, #0x20
  00488C78: F117 0308 adds        r3, r7, #8
  00488C7C: 1D3A      adds        r2, r7, #4
  00488C7E: 1C39      adds        r1, r7, #0
  00488C80: F7FF FFAC bl          target
  00488C84: B001      add         sp, sp, #4
  00488C86: BD00      pop         {pc}
```

`.pdata` (fixed, 2 words):

- Word 0
 - `Function Start RVA` = 0x00088C72 (= 0x00488C72-0x00400000)
- Word 1
 - `Flag` = 1, indicating canonical prologue and epilogue formats
 - `Function Length` = 0x0B (= 0x16/2)
 - `Ret` = 0, indicating a pop {pc} return

- `H` = 0, indicating the parameters weren't homed
- `R` = 0 and `Reg` = 7, indicating no registers were saved/restored
- `L` = 1, indicating LR was saved/restored
- `C` = 0, indicating no frame chaining
- `Stack Adjust` = 1, indicating a 1 × 4 byte stack adjustment

See also

Overview of ARM ABI Conventions
Common Visual C++ ARM Migration Issues

ARM64 exception handling

Article • 03/21/2023

Windows on ARM64 uses the same structured exception handling mechanism for asynchronous hardware-generated exceptions and synchronous software-generated exceptions. Language-specific exception handlers are built on top of Windows structured exception handling by using language helper functions. This document describes exception handling in Windows on ARM64. It illustrates the language helpers used by code that's generated by the Microsoft ARM assembler and the MSVC compiler.

Goals and motivation

The exception unwinding data conventions, and this description, are intended to:

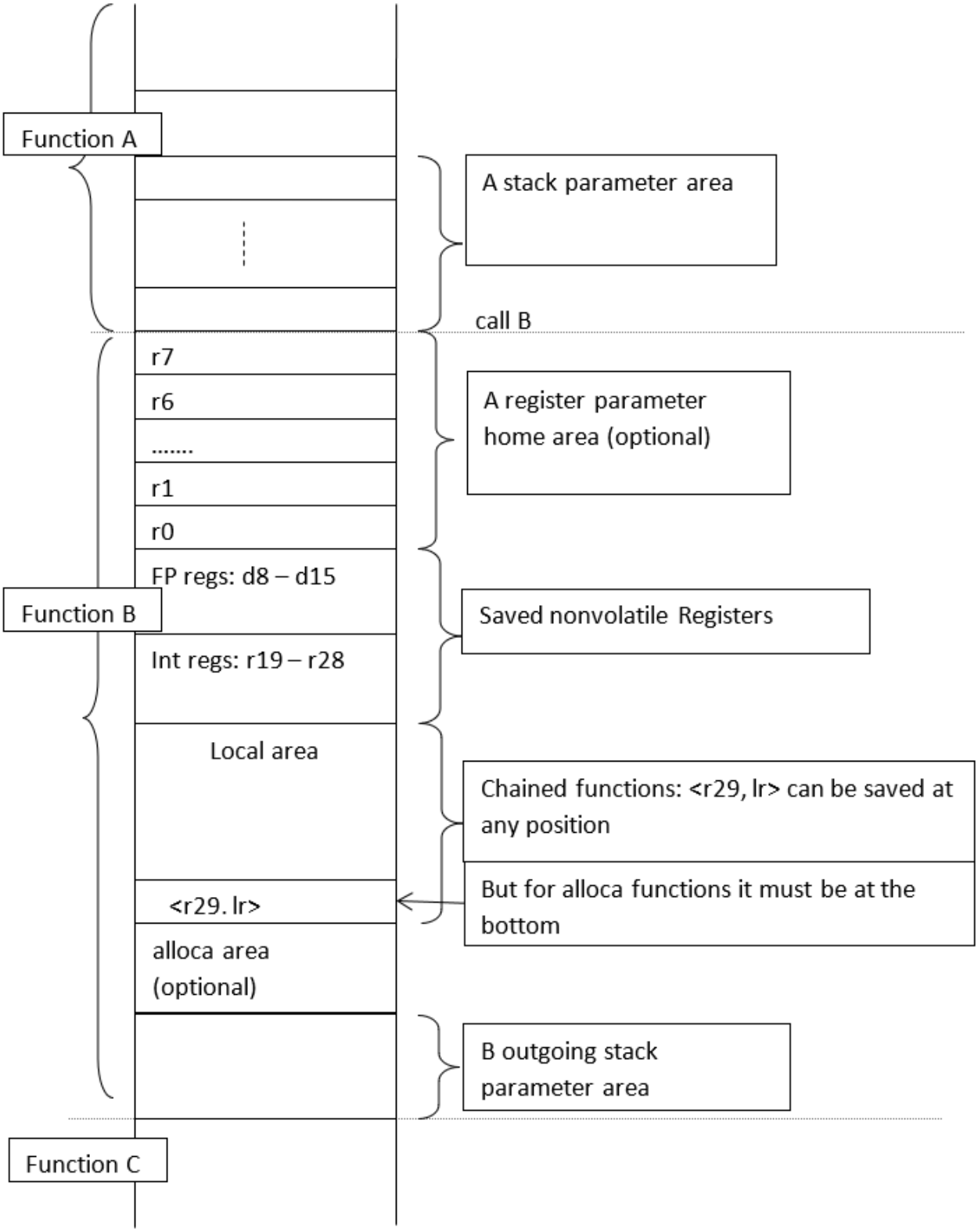
- Provide enough description to allow unwinding without code probing in all cases.
 - Analyzing the code requires the code to be paged in. It prevents unwinding in some circumstances where it's useful (tracing, sampling, debugging).
 - Analyzing the code is complex; the compiler must be careful to only generate instructions that the unwinder can decode.
 - If unwinding can't be fully described by using unwind codes, then in some cases it must fall back to instruction decoding. Instruction decoding increases the overall complexity, and ideally should be avoided.
- Support unwinding in mid-prolog and mid-epilog.
 - Unwinding is used in Windows for more than exception handling. It's critical that code can unwind accurately even when in the middle of a prolog or epilog code sequence.
- Take up a minimal amount of space.
 - The unwind codes must not aggregate to significantly increase the binary size.
 - Since the unwind codes are likely to be locked in memory, a small footprint ensures a minimal overhead for each loaded binary.

Assumptions

These assumptions are made in the exception handling description:

- Prologs and epilogs tend to mirror each other. By taking advantage of this common trait, the size of the metadata needed to describe unwinding can be greatly reduced. Within the body of the function, it doesn't matter whether the prolog's operations are undone, or the epilog's operations are done in a forward manner. Both should produce identical results.
- Functions tend on the whole to be relatively small. Several optimizations for space rely on this fact to achieve the most efficient packing of data.
- There's no conditional code in epilogs.
- Dedicated frame pointer register: If the `sp` is saved in another register (`x29`) in the prolog, that register remains untouched throughout the function. It means the original `sp` may be recovered at any time.
- Unless the `sp` is saved in another register, all manipulation of the stack pointer occurs strictly within the prolog and epilog.
- The stack frame layout is organized as described in the next section.

ARM64 stack frame layout



For frame chained functions, the `fp` and `lr` pair can be saved at any position in the local variable area, depending on optimization considerations. The goal is to maximize the number of locals that can be reached by a single instruction based on the frame pointer (`x29`) or stack pointer (`sp`). However, for `alloca` functions, it must be chained, and `x29` must point to the bottom of stack. To allow for better register-pair-addressing-mode coverage, nonvolatile register save areas are positioned at the top of the Local area stack. Here are examples that illustrate several of the most efficient prolog sequences. For the sake of clarity and better cache locality, the order of storing callee-saved registers in all canonical prologs is in "growing up" order. `#framesz` below

represents the size of entire stack (excluding `alloca` area). `#localsz` and `#outsz` denote local area size (including the save area for the `<x29, lr>` pair) and outgoing parameter size, respectively.

1. Chained, #localsz <= 512

```asm
    stp    x19,x20,[sp,#-96]!        // pre-indexed, save in 1st FP/INT
pair
    stp    d8,d9,[sp,#16]            // save in FP regs (optional)
    stp    x0,x1,[sp,#32]            // home params (optional)
    stp    x2,x3,[sp,#48]
    stp    x4,x5,[sp,#64]
    stp    x6,x7,[sp,#82]
    stp    x29,lr,[sp,#-localsz]!   // save <x29,lr> at bottom of local
area
    mov    x29,sp                   // x29 points to bottom of local
    sub    sp,sp,#outsz             // (optional for #outsz != 0)
```

2. Chained, #localsz > 512

```asm
    stp    x19,x20,[sp,#-96]!        // pre-indexed, save in 1st FP/INT
pair
    stp    d8,d9,[sp,#16]            // save in FP regs (optional)
    stp    x0,x1,[sp,#32]            // home params (optional)
    stp    x2,x3,[sp,#48]
    stp    x4,x5,[sp,#64]
    stp    x6,x7,[sp,#82]
    sub    sp,sp,#(localsz+outsz)   // allocate remaining frame
    stp    x29,lr,[sp,#outsz]       // save <x29,lr> at bottom of local
area
    add    x29,sp,#outsz            // setup x29 points to bottom of
local area
```

3. Unchained, leaf functions (`lr` unsaved)

```asm
    stp    x19,x20,[sp,#-80]!       // pre-indexed, save in 1st FP/INT
reg-pair
    stp    x21,x22,[sp,#16]
    str    x23,[sp,#32]
    stp    d8,d9,[sp,#40]           // save FP regs (optional)
    stp    d10,d11,[sp,#56]
    sub    sp,sp,#(framesz-80)      // allocate the remaining local
area
```

All locals are accessed based on `sp`. `<x29,lr>` points to the previous frame. For frame size <= 512, the `sub sp, ...` can be optimized away if the regs saved area is moved to the bottom of stack. The downside is that it's not consistent with other layouts above. And, saved regs take part of the range for pair-regs and pre- and post-indexed offset addressing mode.

4. Unchained, non-leaf functions (saves `lr` in Int saved area)

```asm
    stp    x19,x20,[sp,#-80]!       // pre-indexed, save in 1st FP/INT
reg-pair
    stp    x21,x22,[sp,#16]         // ...
    stp    x23,lr,[sp,#32]          // save last Int reg and lr
    stp    d8,d9,[sp,#48]           // save FP reg-pair (optional)
    stp    d10,d11,[sp,#64]         // ...
    sub    sp,sp,#(framesz-80)      // allocate the remaining local
area
```

Or, with even number saved Int registers,

```asm
    stp    x19,x20,[sp,#-80]!       // pre-indexed, save in 1st FP/INT
reg-pair
    stp    x21,x22,[sp,#16]         // ...
    str    lr,[sp,#32]              // save lr
    stp    d8,d9,[sp,#40]           // save FP reg-pair (optional)
    stp    d10,d11,[sp,#56]         // ...
    sub    sp,sp,#(framesz-80)      // allocate the remaining local
area
```

Only `x19` saved:

```asm
    sub    sp,sp,#16                // reg save area allocation*
    stp    x19,lr,[sp]              // save x19, lr
    sub    sp,sp,#(framesz-16)      // allocate the remaining local
area
```

* The reg save area allocation isn't folded into the `stp` because a pre-indexed reg-lr `stp` can't be represented with the unwind codes.

All locals are accessed based on `sp`. `<x29>` points to the previous frame.

5. Chained, #framesz <= 512, #outsz = 0

```asm
    stp    x29,lr,[sp,#-framesz]!       // pre-indexed, save <x29,lr>
    mov    x29,sp                       // x29 points to bottom of
stack
    stp    x19,x20,[sp,#(framesz-32)]   // save INT pair
    stp    d8,d9,[sp,#(framesz-16)]     // save FP pair
```

Compared to the first prolog example above, this example has an advantage: all register save instructions are ready to execute after only one stack allocation instruction. That means there's no anti-dependence on `sp` that prevents instruction level parallelism.

6. Chained, frame size > 512 (optional for functions without `alloca`)

```asm
    stp    x29,lr,[sp,#-80]!            // pre-indexed, save <x29,lr>
    stp    x19,x20,[sp,#16]             // save in INT regs
    stp    x21,x22,[sp,#32]             // ...
    stp    d8,d9,[sp,#48]               // save in FP regs
    stp    d10,d11,[sp,#64]
    mov    x29,sp                       // x29 points to top of local
area
    sub    sp,sp,#(framesz-80)          // allocate the remaining local
area
```

For optimization purpose, `x29` can be put at any position in local area to provide a better coverage for "reg-pair" and pre-/post-indexed offset addressing mode. Locals below frame pointers can be accessed based on `sp`.

7. Chained, frame size > 4K, with or without alloca(),

```asm
    stp    x29,lr,[sp,#-80]!            // pre-indexed, save <x29,lr>
    stp    x19,x20,[sp,#16]             // save in INT regs
    stp    x21,x22,[sp,#32]             // ...
    stp    d8,d9,[sp,#48]               // save in FP regs
    stp    d10,d11,[sp,#64]
    mov    x29,sp                       // x29 points to top of local
area
    mov    x15,#(framesz/16)
    bl     __chkstk
    sub    sp,sp,x15,lsl#4              // allocate remaining frame
                                        // end of prolog
    ...
    sub    sp,sp,#alloca                // more alloca() in body
    ...
```

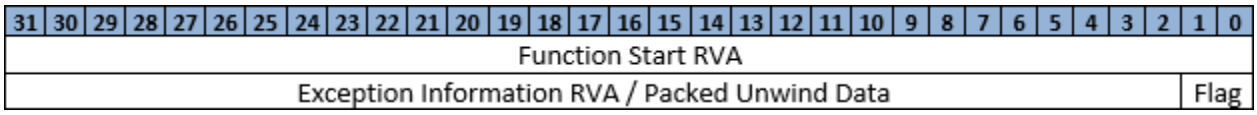
```
                                        // beginning of epilog
    mov    sp,x29                       // sp points to top of local
area
    ldp    d10,d11,[sp,#64]
    ...
    ldp    x29,lr,[sp],#80              // post-indexed, reload
<x29,lr>
```

ARM64 exception handling information

`.pdata` records

The `.pdata` records are an ordered array of fixed-length items that describe every stack-manipulating function in a PE binary. The phrase "stack-manipulating" is significant: leaf functions that don't require any local storage, and don't need to save/restore non-volatile registers, don't require a `.pdata` record. These records should be explicitly omitted to save space. An unwind from one of these functions can get the return address directly from `lr` to move up to the caller.

Each `.pdata` record for ARM64 is 8 bytes in length. The general format of each record places the 32-bit RVA of the function start in the first word, followed by a second word that contains either a pointer to a variable-length `.xdata` block, or a packed word describing a canonical function unwinding sequence.



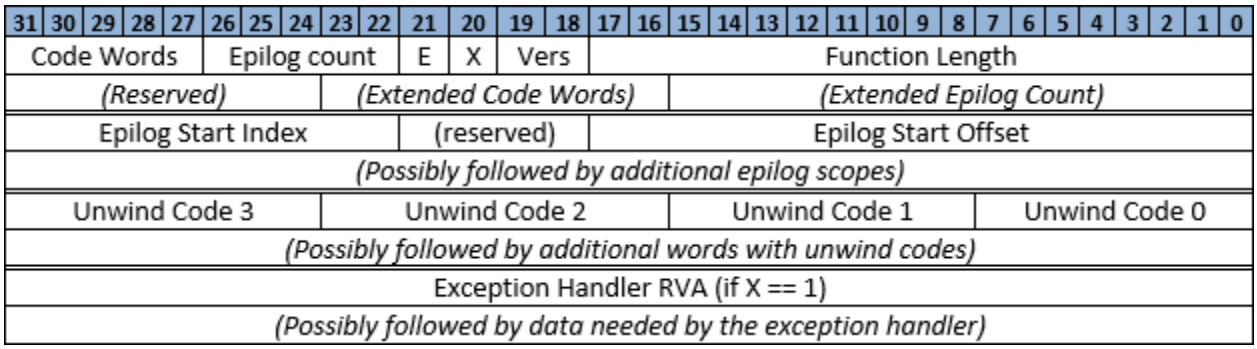
The fields are as follows:

- **Function Start RVA** is the 32-bit RVA of the start of the function.
- **Flag** is a 2-bit field that indicates how to interpret the remaining 30 bits of the second `.pdata` word. If **Flag** is 0, then the remaining bits form an **Exception Information RVA** (with the two lowest bits implicitly 0). If **Flag** is non-zero, then the remaining bits form a **Packed Unwind Data** structure.
- **Exception Information RVA** is the address of the variable-length exception information structure, stored in the `.xdata` section. This data must be 4-byte aligned.
- **Packed Unwind Data** is a compressed description of the operations needed to unwind from a function, assuming a canonical form. In this case, no `.xdata` record

is required.

`.xdata` records

When the packed unwind format is insufficient to describe the unwinding of a function, a variable-length `.xdata` record must be created. The address of this record is stored in the second word of the `.pdata` record. The format of the `.xdata` is a packed variable-length set of words:



This data is broken into four sections:

1. A 1-word or 2-word header describing the overall size of the structure and providing key function data. The second word is only present if both the **Epilog Count** and **Code Words** fields are set to 0. The header has these bit fields:

 a. **Function Length** is an 18-bit field. It indicates the total length of the function in bytes, divided by 4. If a function is larger than 1M, then multiple `.pdata` and `.xdata` records must be used to describe the function. For more information, see the Large functions section.

 b. **Vers** is a 2-bit field. It describes the version of the remaining `.xdata`. Currently, only version 0 is defined, so values of 1-3 aren't permitted.

 c. **X** is a 1-bit field. It indicates the presence (1) or absence (0) of exception data.

 d. **E** is a 1-bit field. It indicates that information describing a single epilog is packed into the header (1) rather than requiring more scope words later (0).

 e. **Epilog Count** is a 5-bit field that has two meanings, depending on the state of **E** bit:

 a. If **E** is 0, it specifies the count of the total number of epilog scopes described in section 2. If more than 31 scopes exist in the function, then the **Code Words** field must be set to 0 to indicate that an extension word is required.

b. If **E** is 1, then this field specifies the index of the first unwind code that describes the one and only epilog.

f. **Code Words** is a 5-bit field that specifies the number of 32-bit words needed to contain all of the unwind codes in section 3. If more than 31 words (that is, 124 unwind codes) are required, then this field must be 0 to indicate that an extension word is required.

g. **Extended Epilog Count** and **Extended Code Words** are 16-bit and 8-bit fields, respectively. They provide more space for encoding an unusually large number of epilogs, or an unusually large number of unwind code words. The extension word that contains these fields is only present if both the **Epilog Count** and **Code Words** fields in the first header word are 0.

2. If the count of epilogs isn't zero, a list of information about epilog scopes, packed one to a word, comes after the header and optional extended header. They're stored in order of increasing starting offset. Each scope contains the following bits:

 a. **Epilog Start Offset** is an 18-bit field that has the offset in bytes, divided by 4, of the epilog relative to the start of the function.

 b. **Res** is a 4-bit field reserved for future expansion. Its value must be 0.

 c. **Epilog Start Index** is a 10-bit field (2 more bits than **Extended Code Words**). It indicates the byte index of the first unwind code that describes this epilog.

3. After the list of epilog scopes comes an array of bytes that contain unwind codes, described in detail in a later section. This array is padded at the end to the nearest full word boundary. Unwind codes are written to this array. They start with the one closest to the body of the function, and move towards the edges of the function. The bytes for each unwind code are stored in big-endian order so the most significant byte gets fetched first, which identifies the operation and the length of the rest of the code.

4. Finally, after the unwind code bytes, if the **X** bit in the header was set to 1, comes the exception handler information. It consists of a single **Exception Handler RVA** that provides the address of the exception handler itself. It's followed immediately by a variable-length amount of data required by the exception handler.

The `.xdata` record is designed so it's possible to fetch the first 8 bytes, and use them to compute the full size of the record, minus the length of the variable-sized exception data that follows. The following code snippet computes the record size:

```
C++
```

```
ULONG ComputeXdataSize(PULONG Xdata)
{
    ULONG Size;
    ULONG EpilogScopes;
    ULONG UnwindWords;

    if ((Xdata[0] >> 22) != 0) {
        Size = 4;
        EpilogScopes = (Xdata[0] >> 22) & 0x1f;
        UnwindWords = (Xdata[0] >> 27) & 0x1f;
    } else {
        Size = 8;
        EpilogScopes = Xdata[1] & 0xffff;
        UnwindWords = (Xdata[1] >> 16) & 0xff;
    }

    if (!(Xdata[0] & (1 << 21))) {
        Size += 4 * EpilogScopes;
    }

    Size += 4 * UnwindWords;

    if (Xdata[0] & (1 << 20)) {
        Size += 4;  // Exception handler RVA
    }

    return Size;
}
```

Although the prolog and each epilog has its own index into the unwind codes, the table is shared between them. It's entirely possible (and not altogether uncommon) that they can all share the same codes. (For an example, see Example 2 in the Examples section.) Compiler writers should optimize for this case in particular. It's because the largest index that can be specified is 255, which limits the total number of unwind codes for a particular function.

Unwind codes

The array of unwind codes is a pool of sequences that describe exactly how to undo the effects of the prolog. They're stored in the same order the operations need to be undone. The unwind codes can be thought of as a small instruction set, encoded as a string of bytes. When execution is complete, the return address to the calling function is in the `lr` register. And, all non-volatile registers are restored to their values at the time the function was called.

If exceptions were guaranteed to only ever occur within a function body, and never within a prolog or any epilog, then only a single sequence would be necessary. However,

the Windows unwinding model requires that code can unwind from within a partially executed prolog or epilog. To meet this requirement, the unwind codes have been carefully designed so they unambiguously map 1:1 to each relevant opcode in the prolog and epilog. This design has several implications:

- By counting the number of unwind codes, it's possible to compute the length of the prolog and epilog.
- By counting the number of instructions past the start of an epilog scope, it's possible to skip the equivalent number of unwind codes. We can execute the rest of a sequence to complete the partially executed unwind done by the epilog.
- By counting the number of instructions before the end of the prolog, it's possible to skip the equivalent number of unwind codes. We can execute the rest of the sequence to undo only those parts of the prolog that have completed execution.

The unwind codes are encoded according to the table below. All unwind codes are a single/double byte, except the one that allocates a huge stack (`alloc_l`). There are 22 unwind codes in total. Each unwind code maps exactly one instruction in the prolog/epilog, to allow for unwinding of partially executed prologs and epilogs.

Unwind code	Bits and interpretation
`alloc_s`	000xxxxx: allocate small stack with size < 512 (2^5 * 16).
`save_r19r20_x`	001zzzzz: save `<x19,x20>` pair at `[sp-#Z*8]!`, pre-indexed offset >= -248
`save_fplr`	01zzzzzz: save `<x29,lr>` pair at `[sp+#Z*8]`, offset <= 504.
`save_fplr_x`	10zzzzzz: save `<x29,lr>` pair at `[sp-(#Z+1)*8]!`, pre-indexed offset >= -512
`alloc_m`	11000xxx'xxxxxxxx: allocate large stack with size < 32K (2^11 * 16).
`save_regp`	110010xx'xxzzzzzz: save `x(19+#X)` pair at `[sp+#Z*8]`, offset <= 504
`save_regp_x`	110011xx'xxzzzzzz: save pair `x(19+#X)` at `[sp-(#Z+1)*8]!`, pre-indexed offset >= -512
`save_reg`	110100xx'xxzzzzzz: save reg `x(19+#X)` at `[sp+#Z*8]`, offset <= 504
`save_reg_x`	1101010x'xxxzzzzz: save reg `x(19+#X)` at `[sp-(#Z+1)*8]!`, pre-indexed offset >= -256
`save_lrpair`	1101011x'xxzzzzzz: save pair `<x(19+2*#X),lr>` at `[sp+#Z*8]`, offset <= 504
`save_fregp`	1101100x'xxzzzzzz: save pair `d(8+#X)` at `[sp+#Z*8]`, offset <= 504

Unwind code	Bits and interpretation
`save_fregp_x`	1101101x'xxzzzzzz: save pair `d(8+#X)` at `[sp-(#Z+1)*8]!`, pre-indexed offset >= -512
`save_freg`	1101110x'xxzzzzzz: save reg `d(8+#X)` at `[sp+#Z*8]`, offset <= 504
`save_freg_x`	11011110'xxxzzzzz: save reg `d(8+#X)` at `[sp-(#Z+1)*8]!`, pre-indexed offset >= -256
`alloc_l`	11100000'xxxxxxxx'xxxxxxxx'xxxxxxxx: allocate large stack with size < 256M (2^24 * 16)
`set_fp`	11100001: set up `x29` with `mov x29,sp`
`add_fp`	11100010'xxxxxxxx: set up `x29` with `add x29,sp,#x*8`
`nop`	11100011: no unwind operation is required.
`end`	11100100: end of unwind code. Implies `ret` in epilog.
`end_c`	11100101: end of unwind code in current chained scope.
`save_next`	11100110: save next non-volatile Int or FP register pair.
	11100111: reserved
	11101xxx: reserved for custom stack cases below only generated for asm routines
	11101000: Custom stack for `MSFT_OP_TRAP_FRAME`
	11101001: Custom stack for `MSFT_OP_MACHINE_FRAME`
	11101010: Custom stack for `MSFT_OP_CONTEXT`
	11101011: Custom stack for `MSFT_OP_EC_CONTEXT`
	11101100: Custom stack for `MSFT_OP_CLEAR_UNWOUND_TO_CALL`
	11101101: reserved
	11101110: reserved
	11101111: reserved
	11110xxx: reserved
	11111000'yyyyyyyy : reserved
	11111001'yyyyyyyy'yyyyyyyy : reserved
	11111010'yyyyyyyy'yyyyyyyy'yyyyyyyy : reserved

Unwind code	Bits and interpretation
	11111011'yyyyyyyy'yyyyyyyy'yyyyyyyy'yyyyyyyy : reserved
`pac_sign_lr`	11111100: sign the return address in `lr` with `pacibsp`
	11111101: reserved
	11111110: reserved
	11111111: reserved

In instructions with large values covering multiple bytes, the most significant bits are stored first. This design makes it possible to find the total size in bytes of the unwind code by looking up only the first byte of the code. Since each unwind code is exactly mapped to an instruction in a prolog or epilog, you can compute the size of the prolog or epilog. Walk from the sequence start to the end, and use a lookup table or similar device to determine the length of the corresponding opcode.

Post-indexed offset addressing isn't allowed in a prolog. All offset ranges (#Z) match the encoding of `stp`/`str` addressing except `save_r19r20_x`, in which 248 is sufficient for all save areas (10 Int registers + 8 FP registers + 8 input registers).

`save_next` must follow a save for Int or FP volatile register pair: `save_regp`, `save_regp_x`, `save_fregp`, `save_fregp_x`, `save_r19r20_x`, or another `save_next`. It saves the next register pair at the next 16-byte slot in "growing up" order. A `save_next` refers to the first FP register pair when it follows the `save-next` that denotes the last Int register pair.

Since the sizes of regular return and jump instructions are the same, there's no need for a separated `end` unwind code in tail-call scenarios.

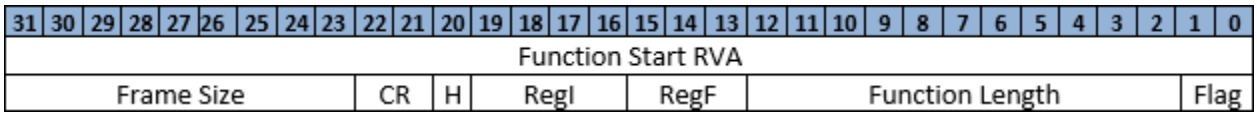
`end_c` is designed to handle noncontiguous function fragments for optimization purposes. An `end_c` that indicates the end of unwind codes in the current scope must be followed by another series of unwind codes ending with a real `end`. The unwind codes between `end_c` and `end` represent the prolog operations in the parent region (a "phantom" prolog). More details and examples are described in the section below.

Packed unwind data

For functions whose prologs and epilogs follow the canonical form described below, packed unwind data can be used. It eliminates the need for an `.xdata` record entirely, and significantly reduces the cost of providing unwind data. The canonical prologs and epilogs are designed to meet the common requirements of a simple function: One that

doesn't require an exception handler, and which does its setup and teardown operations in a standard order.

The format of a `.pdata` record with packed unwind data looks like this:



The fields are as follows:

- **Function Start RVA** is the 32-bit RVA of the start of the function.
- **Flag** is a 2-bit field as described above, with the following meanings:
 - 00 = packed unwind data not used; remaining bits point to an `.xdata` record
 - 01 = packed unwind data used with a single prolog and epilog at the beginning and end of the scope
 - 10 = packed unwind data used for code without any prolog and epilog. Useful for describing separated function segments
 - 11 = reserved.
- **Function Length** is an 11-bit field providing the length of the entire function in bytes, divided by 4. If the function is larger than 8k, a full `.xdata` record must be used instead.
- **Frame Size** is a 9-bit field indicating the number of bytes of stack that is allocated for this function, divided by 16. Functions that allocate greater than (8k-16) bytes of stack must use a full `.xdata` record. It includes the local variable area, outgoing parameter area, callee-saved Int and FP area, and home parameter area. It excludes the dynamic allocation area.
- **CR** is a 2-bit flag indicating whether the function includes extra instructions to set up a frame chain and return link:
 - 00 = unchained function, `<x29,lr>` pair isn't saved in stack
 - 01 = unchained function, `<lr>` is saved in stack
 - 10 = chained function with a `pacibsp` signed return address
 - 11 = chained function, a store/load pair instruction is used in prolog/epilog `<x29,lr>`
- **H** is a 1-bit flag indicating whether the function homes the integer parameter registers (x0-x7) by storing them at the very start of the function. (0 = doesn't home registers, 1 = homes registers).
- **RegI** is a 4-bit field indicating the number of non-volatile INT registers (x19-x28) saved in the canonical stack location.
- **RegF** is a 3-bit field indicating the number of non-volatile FP registers (d8-d15) saved in the canonical stack location. (RegF=0: no FP register is saved; RegF>0:

RegF+1 FP registers are saved). Packed unwind data can't be used for function that save only one FP register.

Canonical prologs that fall into categories 1, 2 (without outgoing parameter area), 3 and 4 in section above can be represented by packed unwind format. The epilogs for canonical functions follow a similar form, except **H** has no effect, the `set_fp` instruction is omitted, and the order of steps and the instructions in each step are reversed in the epilog. The algorithm for packed `.xdata` follows these steps, detailed in the following table:

Step 0: Pre-compute of the size of each area.

Step 1: Sign the return address.

Step 2: Save Int callee-saved registers.

Step 3: This step is specific for type 4 in early sections. `lr` is saved at the end of Int area.

Step 4: Save FP callee-saved registers.

Step 5: Save input arguments in the home parameter area.

Step 6: Allocate remaining stack, including local area, `<x29,lr>` pair, and outgoing parameter area. 6a corresponds to canonical type 1. 6b and 6c are for canonical type 2. 6d and 6e are for both type 3 and type 4.

Step #	**Flag values**	**# of instructions**	**Opcode**	**Unwind code**
0			`#intsz = RegI * 8;` `if (CR==01) #intsz += 8; // lr` `#fpsz = RegF * 8;` `if(RegF) #fpsz += 8;` `#savsz=` `((#intsz+#fpsz+8*8*H)+0xf)&~0xf)` `#locsz = #famsz - #savsz`	
1	**CR** == 10	1	`pacibsp`	`pac_sign_lr`
2	0 < **RegI** <= 10	**RegI** / 2 + **RegI** % 2	`stp x19,x20,[sp,#savsz]!` `stp x21,x22,[sp,#16]` `...`	`save_regp_x` `save_regp` `...`
3	**CR** == 01*	1	`str lr,[sp,#(intsz-8)]`*	`save_reg`

Step #	Flag values	# of instructions	Opcode	Unwind code
4	0 < **RegF** <= 7	(**RegF** + 1) / 2 + (**RegF** + 1) % 2)	`stp d8,d9,[sp,#intsz]`** `stp d10,d11,[sp,#(intsz+16)]` `...` `str d(8+RegF),[sp,#(intsz+fpsz-8)]`	`save_fregp` `...` `save_freg`
5	**H** == 1	4	`stp x0,x1,[sp,#(intsz+fpsz)]` `stp x2,x3,[sp,#(intsz+fpsz+16)]` `stp x4,x5,[sp,#(intsz+fpsz+32)]` `stp x6,x7,[sp,#(intsz+fpsz+48)]`	`nop` `nop` `nop` `nop`
6a	(**CR** == 10 \|\| **CR** == 11) && `#locsz` <= 512	2	`stp x29,lr,[sp,#-locsz]!` `mov x29,sp`***	`save_fplr_x` `set_fp`
6b	(**CR** == 10 \|\| **CR** == 11) && 512 < `#locsz` <= 4080	3	`sub sp,sp,#locsz` `stp x29,lr,[sp,0]` `add x29,sp,0`	`alloc_m` `save_fplr` `set_fp`
6c	(**CR** == 10 \|\| **CR** == 11) && `#locsz` > 4080	4	`sub sp,sp,4080` `sub sp,sp,#(locsz-4080)` `stp x29,lr,[sp,0]` `add x29,sp,0`	`alloc_m` `alloc_s`/`alloc_m` `save_fplr` `set_fp`
6d	(**CR** == 00 \|\| **CR** == 01) && `#locsz` <= 4080	1	`sub sp,sp,#locsz`	`alloc_s`/`alloc_m`
6e	(**CR** == 00 \|\| **CR** == 01) && `#locsz` > 4080	2	`sub sp,sp,4080` `sub sp,sp,#(locsz-4080)`	`alloc_m` `alloc_s`/`alloc_m`

* If **CR** == 01 and **RegI** is an odd number, Step 2 and the last `save_rep` in step 1 are merged into one `save_regp`.

** If **RegI** == **CR** == 0, and **RegF** != 0, the first `stp` for the floating-point does the predecrement.

*** No instruction corresponding to `mov x29,sp` is present in the epilog. Packed unwind data can't be used if a function requires restoration of `sp` from `x29`.

Unwinding partial prologs and epilogs

In the most common unwinding situations, the exception or call occurs in the body of the function, away from the prolog and all epilogs. In these situations, unwinding is straightforward: the unwinder simply executes the codes in the unwind array. It begins at index 0 and continues until an `end` opcode is detected.

It's more difficult to correctly unwind in the case where an exception or interrupt occurs while executing a prolog or epilog. In these situations, the stack frame is only partially constructed. The problem is to determine exactly what's been done, to correctly undo it.

For example, take this prolog and epilog sequence:

asm

```
0000:    stp    x29,lr,[sp,#-256]!          // save_fplr_x  256 (pre-indexed
store)
0004:    stp    d8,d9,[sp,#224]             // save_fregp 0, 224
0008:    stp    x19,x20,[sp,#240]           // save_regp 0, 240
000c:    mov    x29,sp                      // set_fp
         ...
0100:    mov    sp,x29                      // set_fp
0104:    ldp    x19,x20,[sp,#240]           // save_regp 0, 240
0108:    ldp    d8,d9,[sp,224]              // save_fregp 0, 224
010c:    ldp    x29,lr,[sp],#256            // save_fplr_x  256 (post-
indexed load)
0110:    ret    lr                          // end
```

Next to each opcode is the appropriate unwind code describing this operation. You can see how the series of unwind codes for the prolog is an exact mirror image of the unwind codes for the epilog (not counting the final instruction of the epilog). It's a common situation: It's why we always assume the unwind codes for the prolog are stored in reverse order from the prolog's execution order.

So, for both the prolog and epilog, we're left with a common set of unwind codes:

`set_fp`, `save_regp 0,240`, `save_fregp,0,224`, `save_fplr_x_256`, `end`

The epilog case is straightforward, since it's in normal order. Starting at offset 0 within the epilog (which starts at offset 0x100 in the function), we'd expect the full unwind sequence to execute, as no cleanup has yet been done. If we find ourselves one instruction in (at offset 2 in the epilog), we can successfully unwind by skipping the first unwind code. We can generalize this situation, and assume a 1:1 mapping between opcodes and unwind codes. Then, to start unwinding from instruction *n* in the epilog, we should skip the first *n* unwind codes, and begin executing from there.

It turns out that a similar logic works for the prolog, except in reverse. If we start unwinding from offset 0 in the prolog, we want to execute nothing. If we unwind from

offset 2, which is one instruction in, then we want to start executing the unwind sequence one unwind code from the end. (Remember, the codes are stored in reverse order.) And here too, we can generalize: if we start unwinding from instruction *n* in the prolog, we should start executing *n* unwind codes from the end of the list of codes.

Prolog and epilog codes don't always match exactly, which is why the unwind array may need to contain several sequences of codes. To determine the offset of where to begin processing codes, use the following logic:

1. If unwinding from within the body of the function, begin executing unwind codes at index 0 and continue until hitting an `end` opcode.

2. If unwinding from within an epilog, use the epilog-specific starting index provided with the epilog scope as a starting point. Compute how many bytes the PC in question is from the start of the epilog. Then advance forward through the unwind codes, skipping unwind codes until all of the already-executed instructions are accounted for. Then execute starting at that point.

3. If unwinding from within the prolog, use index 0 as your starting point. Compute the length of the prolog code from the sequence, and then compute how many bytes the PC in question is from the end of the prolog. Then advance forward through the unwind codes, skipping unwind codes until all of the not-yet-executed instructions are accounted for. Then execute starting at that point.

These rules mean the unwind codes for the prolog must always be the first in the array. And, they're also the codes used to unwind in the general case of unwinding from within the body. Any epilog-specific code sequences should follow immediately after.

Function fragments

For code optimization purposes and other reasons, it may be preferable to split a function into separated fragments (also called *regions*). When split, each resulting function fragment requires its own separate `.pdata` (and possibly `.xdata`) record.

For each separated secondary fragment that has its own prolog, it's expected that no stack adjustment is done in its prolog. All stack space required by a secondary region must be pre-allocated by its parent region (or called host region). This preallocation keeps stack pointer manipulation strictly in the function's original prolog.

A typical case of function fragments is "code separation", where the compiler may move a region of code out of its host function. There are three unusual cases that could result from code separation.

Example

- (region 1: begin)

```asm
    stp     x29,lr,[sp,#-256]!      // save_fplr_x  256 (pre-indexed
store)
    stp     x19,x20,[sp,#240]       // save_regp 0, 240
    mov     x29,sp                  // set_fp
    ...
```

- (region 1: end)
- (region 3: begin)

```asm
    ...
```

- (region 3: end)
- (region 2: begin)

```asm
    ...
    mov     sp,x29                  // set_fp
    ldp     x19,x20,[sp,#240]       // save_regp 0, 240
    ldp     x29,lr,[sp],#256        // save_fplr_x  256 (post-indexed
load)
    ret     lr                      // end
```

- (region 2: end)

1. Prolog only (region 1: all epilogs are in separated regions):

 Only the prolog must be described. This prolog can't be represented in the compact `.pdata` format. In the full `.xdata` case, it can be represented by setting Epilog Count = 0. See region 1 in the example above.

 Unwind codes: `set_fp`, `save_regp 0,240`, `save_fplr_x_256`, `end`.

2. Epilogs only (region 2: prolog is in host region)

 It's assumed that by the time control jumps into this region, all prolog codes have been executed. Partial unwind can happen in epilogs the same way as in a normal

function. This type of region can't be represented by compact `.pdata`. In a full `.xdata` record, it can be encoded with a "phantom" prolog, bracketed by an `end_c` and `end` unwind code pair. The leading `end_c` indicates the size of prolog is zero. Epilog start index of the single epilog points to `set_fp`.

Unwind code for region 2: `end_c`, `set_fp`, `save_regp 0,240`, `save_fplr_x_256`, `end`.

3. No prologs or epilogs (region 3: prologs and all epilogs are in other fragments):

 Compact `.pdata` format can be applied via setting Flag = 10. With full `.xdata` record, Epilog Count = 1. Unwind code is the same as the code for region 2 above, but Epilog Start Index also points to `end_c`. Partial unwind will never happen in this region of code.

Another more complicated case of function fragments is "shrink wrapping." The compiler may choose to delay saving some callee-saved registers until outside of the function entry prolog.

- (region 1: begin)

```asm
    stp     x29,lr,[sp,#-256]!      // save_fplr_x  256 (pre-indexed
store)
    stp     x19,x20,[sp,#240]       // save_regp 0, 240
    mov     x29,sp                  // set_fp
    ...
```

- (region 2: begin)

```asm
    stp     x21,x22,[sp,#224]       // save_regp 2, 224
    ...
    ldp     x21,x22,[sp,#224]       // save_regp 2, 224
```

- (region 2: end)

```asm
    ...
    mov     sp,x29                  // set_fp
    ldp     x19,x20,[sp,#240]       // save_regp 0, 240
    ldp     x29,lr,[sp],#256        // save_fplr_x  256 (post-indexed
load)
    ret     lr                      // end
```

- (region 1: end)

In the prolog of region 1, stack space is pre-allocated. You can see that region 2 will have the same unwind code even it's moved out of its host function.

Region 1: `set_fp`, `save_regp 0,240`, `save_fplr_x_256`, `end`. Epilog Start Index points to `set_fp` as usual.

Region 2: `save_regp 2, 224`, `end_c`, `set_fp`, `save_regp 0,240`, `save_fplr_x_256`, `end`. Epilog Start Index points to first unwind code `save_regp 2, 224`.

Large functions

Fragments can be used to describe functions larger than the 1M limit imposed by the bit fields in the `.xdata` header. To describe an unusually large function like this, it needs to be broken into fragments smaller than 1M. Each fragment should be adjusted so that it doesn't split an epilog into multiple pieces.

Only the first fragment of the function will contain a prolog; all other fragments are marked as having no prolog. Depending on the number of epilogs present, each fragment may contain zero or more epilogs. Keep in mind that each epilog scope in a fragment specifies its starting offset relative to the start of the fragment, not the start of the function.

If a fragment has no prolog and no epilog, it still requires its own `.pdata` (and possibly `.xdata`) record, to describe how to unwind from within the body of the function.

Examples

Example 1: Frame-chained, compact-form

asm

```
|Foo|     PROC
|$LN19|
    str     x19,[sp,#-0x10]!        // save_reg_x
    sub     sp,sp,#0x810            // alloc_m
    stp     fp,lr,[sp]              // save_fplr
    mov     fp,sp                   // set_fp
                                    // end of prolog
    ...

|$pdata$Foo|
```

```
    DCD     imagerel     |$LN19|
    DCD     0x416101ed
    ;Flags[SingleProEpi] functionLength[492] RegF[0] RegI[1] H[0]
frameChainReturn[Chained] frameSize[2080]
```

Example 2: Frame-chained, full-form with mirror Prolog & Epilog

asm

```
|Bar|     PROC
|$LN19|
    stp     x19,x20,[sp,#-0x10]!    // save_regp_x
    stp     fp,lr,[sp,#-0x90]!      // save_fplr_x
    mov     fp,sp                   // set_fp
                                    // end of prolog
    ...
                                    // begin of epilog, a mirror sequence of
Prolog
    mov     sp,fp
    ldp     fp,lr,[sp],#0x90
    ldp     x19,x20,[sp],#0x10
    ret     lr

|$pdata$Bar|
    DCD     imagerel     |$LN19|
    DCD     imagerel     |$unwind$cse2|
|$unwind$Bar|
    DCD     0x1040003d
    DCD     0x1000038
    DCD     0xe42291e1
    DCD     0xe42291e1
    ;Code Words[2], Epilog Count[1], E[0], X[0], Function Length[6660]
    ;Epilog Start Index[0], Epilog Start Offset[56]
    ;set_fp
    ;save_fplr_x
    ;save_r19r20_x
    ;end
```

Epilog Start Index [0] points to the same sequence of Prolog unwind code.

Example 3: Variadic unchained Function

asm

```
|Delegate| PROC
|$LN4|
    sub     sp,sp,#0x50
    stp     x19,lr,[sp]
```

```
    stp     x0,x1,[sp,#0x10]        // save incoming register to home area
    stp     x2,x3,[sp,#0x20]        // ...
    stp     x4,x5,[sp,#0x30]
    stp     x6,x7,[sp,#0x40]        // end of prolog
    ...
    ldp     x19,lr,[sp]             // beginning of epilog
    add     sp,sp,#0x50
    ret     lr

    AREA    |.pdata|, PDATA
|$pdata$Delegate|
    DCD     imagerel |$LN4|
    DCD     imagerel |$unwind$Delegate|

    AREA    |.xdata|, DATA
|$unwind$Delegate|
    DCD     0x18400012
    DCD     0x200000f
    DCD     0xe3e3e3e3
    DCD     0xe40500d6
    DCD     0xe40500d6
    ;Code Words[3], Epilog Count[1], E[0], X[0], Function Length[18]
    ;Epilog Start Index[4], Epilog Start Offset[15]
    ;nop        // nop for saving in home area
    ;nop        // ditto
    ;nop        // ditto
    ;nop        // ditto
    ;save_lrpair
    ;alloc_s
    ;end
```

Epilog Start Index [4] points to the middle of Prolog unwind code (partially reuse unwind array).

See also

Overview of ARM64 ABI conventions
ARM exception handling

Configuring Programs for Windows XP

Article • 02/17/2022

Visual Studio supports multiple platform toolsets. That means it's possible to target operating systems and runtime libraries that aren't supported by the default toolset. For example, by switching the platform toolset, you can use the Visual Studio 2017 C++ compiler to create apps that target Windows XP and Windows Server 2003. You can also use older platform toolsets to maintain binary-compatible legacy code and still take advantage of the latest features of the Visual Studio IDE.

The toolset supplied in Visual Studio 2019 and later doesn't include support for creating code for Windows XP. Support for Windows XP development is available by using the Visual Studio 2017 v141_xp toolset. You can install the v141_xp toolset as an individual component option in the Visual Studio Installer.

Install the Windows XP platform toolset

To get the v141_xp platform toolset and components to target Windows XP and Windows Server 2003, run the Visual Studio Installer. When you initially install Visual Studio, or when you modify an existing installation, make sure the **Desktop development with C++** workload is selected. In the **Individual components** tab, under **Compilers, build tools, and runtimes**, choose **C++ Windows XP Support for VS 2017 (v141) tools [Deprecated]**, and then choose **Install** or **Modify**.

Windows XP targeting experience

The Windows XP platform toolset that's included in Visual Studio is a version of the Windows 7 SDK, but it uses the Visual Studio 2017 C++ compiler. It also configures project properties to appropriate default values, for example, the specification of a compatible linker for down-level targeting. Only Windows desktop apps created by using a Windows XP platform toolset can run on Windows XP and Windows Server 2003. Those apps can also run on more recent Windows operating systems.

To target Windows XP

1. In **Solution Explorer**, open the shortcut menu for your project, and then choose **Properties**.

2. In the **Property Pages** dialog box for the project, set the **Configuration** dropdown to **All configurations**.

3. Select the **Configuration Properties** > **General** property page. Set the **Platform Toolset** property to your preferred Windows XP toolset. For example, choose **Visual Studio 2017 - Windows XP (v141_xp)** to create code for Windows XP and Windows Server 2003 by using the Microsoft C++ compiler from Visual Studio 2017.

C++ runtime support

Along with the Windows XP platform toolset, several libraries include runtime support for Windows XP and Windows Server 2003:

- Universal C Runtime Library (UCRT)
- C++ Standard Library
- Active Template Library (ATL)
- Concurrency Runtime Library (ConcRT)
- Parallel Patterns Library (PPL)
- Microsoft Foundation Class Library (MFC)
- C++ AMP (C++ Accelerated Massive Programming) library.

The minimum supported versions of these operating systems are: Windows XP Service Pack 3 (SP3) for x86, Windows XP Service Pack 2 (SP2) for x64, and Windows Server 2003 Service Pack 2 (SP2) for both x86 and x64.

These libraries are supported by the platform toolsets installed by Visual Studio, depending on the target:

Library	**Default platform toolset targeting Windows desktop apps**	**Default platform toolset targeting Store apps**	**Windows XP platform toolset targeting Windows XP, Windows Server 2003**
CRT	X	X	X
C++ Standard Library	X	X	X
ATL	X	X	X
ConcRT/PPL	X	X	X
MFC	X		X
C++ AMP	X	X	

> **Note**

Apps that are written in C++/CLI and target the .NET Framework 4 run on Windows XP and Windows Server 2003.

Differences between the toolsets

Because of differences in platform and library support, the development experience for apps that use a Windows XP platform toolset isn't as complete as for apps that use the default platform toolset.

- **C++ language features**

 Only C++ language features implemented in Visual Studio 2017 are supported in apps that use the v141_xp platform toolset. Only C++ language features implemented in Visual Studio 2015 are supported in apps that use the v140_xp platform toolset. Visual Studio uses the corresponding compiler when it builds using the older platform toolsets. Use the most recent Windows XP platform toolset to take advantage of the latest C++ language features implemented in that version of the compiler. For more information about language feature support by compiler version, see Microsoft C/C++ language conformance.

- **Remote debugging**

 Remote debugging on Windows XP or Windows Server 2003 isn't supported by Remote Tools for Visual Studio. To debug an app locally or remotely on Windows XP or Windows Server 2003, use a debugger from an older version of Visual Studio. It's similar to debugging an app on Windows Vista: Vista is a *runtime* target of the platform toolset, but not a *remote debugging* target.

- **Static analysis**

 The Windows XP platform toolsets don't support static analysis. The SAL annotations for the Windows 7 SDK and the runtime libraries are incompatible. You can still run static analysis on an app that supports Windows XP or Windows Server 2003. Temporarily switch the solution to target the default platform toolset for the analysis, and then switch back to the Windows XP platform toolset to build the app.

- **Debugging of DirectX graphics**

 The Graphics Debugger doesn't support the Direct3D 9 API. It can't be used to debug apps that use Direct3D on Windows XP or Windows Server 2003. However, if the app implements an alternative renderer based on Direct3D 10 or Direct3D 11 APIs, you can use the Graphics Debugger to diagnose problems.

- **Building HLSL**

 The Windows XP toolset doesn't compile HLSL source code files by default. To compile HLSL files, download and install the June 2010 DirectX SDK, and then set the project's VC directories to include it. For more information, see the "DirectX SDK Does Not Register Include/Library Paths with Visual Studio 2010" section of the June 2010 DirectX SDK download page (Archived link).

Windows XP deployment

> **Important**
>
> Because it lacks support for SHA-256 code signing certificates, runtime library support for Windows XP is no longer available in the **latest Visual C++ Redistributable** for Visual Studio 2015, 2017, 2019, and 2022. The last Redistributable to support Windows XP shipped in Visual Studio 2019 version 16.7. Use a Redistributable that has a file version starting with **14.27**. If your Windows XP apps are deployed with or updated to a later version of the redistributable, the apps won't run.

If you're using a version of Visual Studio later than Visual Studio 2019 version 16.7, the redistributable files won't work on Windows XP. To get a copy of the redistributable files that support Windows XP, you'll need a Visual Studio account. Use the account you use to sign in to Visual Studio. Or, you can create an account for free at my.visualstudio.com. The redistributable file is available in the Downloads section, as Visual C++ Redistributable for Visual Studio 2019 - Version 16.7. To download the files, select the platform and language you need, and then choose the Download button.

You can use central deployment or local deployment to install runtime library support for your Windows XP app. For more information, see Walkthrough: Deploying a Visual C++ Application By Using the Visual C++ Redistributable Package.

C++ Code analysis in Visual Studio

Visual Studio provides tools to analyze and improve C++ code quality.

Analyze C and C++ code

OVERVIEW

Code analysis for C/C++ overview

QUICKSTART

Code Analysis for C/C++ quick start

TUTORIAL

Analyze C/C++ code for defects walkthrough

Code analysis reference

OVERVIEW

C++ Core Guidelines

REFERENCE

C++ Core Guidelines warnings

C++ code analysis warnings

Use SAL annotations to reduce defects

OVERVIEW

Understanding SAL

GET STARTED

SAL examples

HOW-TO GUIDE

Annotate function parameters and return values

Annotate function behavior

Annotate structs and classes

Annotate locking behavior

Specify when and where annotations apply

REFERENCE

SAL annotation intrinsics

C/C++ Sanitizers | Instrument code for runtime bug detection

Use C and C++ sanitizers for defect reporting, analysis, and prevention.

Find bugs using code sanitizers

OVERVIEW

Learn about AddressSanitizer

TUTORIAL

AddressSanitizer shadow bytes

AddressSanitizer cloud or distributed testing

Instrument your builds

REFERENCE

AddressSanitizer build and language reference

AddressSanitizer runtime reference

AddressSanitizer error examples

AddressSanitizer known issues

Debug your results

REFERENCE

AddressSanitizer debugger integration

C/C++ Building Reference

Article • 08/03/2021

Visual Studio provides two ways of building a C/C++ program. The easiest (and most common) way is to build within the Visual Studio IDE. The other way is to build from a command prompt using command-line tools. In either case, you can create and edit your source files using Visual Studio or a third-party editor of your choice.

In This Section

MSBuild reference for C++ projects

MSVC Compiler Reference
Describes the MSVC compiler, which creates an object file containing machine code, linker directives, sections, external references, and function/data names.

MSVC linker reference
Describes the linker, which combines code from the object files created by the compiler and from statically linked libraries, resolves the name references, and creates an executable file.

Unicode Support in the Compiler and Linker

Additional MSVC Build Tools
Additional command-line tools for C++.

C/C++ Build Errors
Introduces the build errors section in the table of contents.

Related Sections

C/C++ Preprocessor Reference
Discusses the preprocessor, which prepares source files for the compiler by translating macros, operators, and directives.

Understanding Custom Build Steps and Build Events
Discusses customizing the build process.

Building a C/C++ Program
Provides links to topics describing building your program from the command line or from the integrated development environment of Visual Studio.

MSVC Compiler Command-Line Syntax

Describes setting compiler options in the development environment or on the command line.

MSVC Compiler Options

Provides links to topics discussing using compiler options.

MSVC linker reference

Describes setting linker options inside or outside the integrated development environment.

MSVC Linker Options

Provides links to topics discussing using linker options.

BSCMAKE Reference

Describes the Microsoft Browse Information Maintenance Utility (BSCMAKE.EXE), which builds a browse information file (.bsc) from .sbr files created during compilation.

LIB Reference

Describes the Microsoft Library Manager (LIB.exe), which creates and manages a library of Common Object File Format (COFF) object files.

EDITBIN Reference

Describes the Microsoft COFF Binary File Editor (EDITBIN.EXE), which modifies Common Object File Format (COFF) binary files.

DUMPBIN Reference

Describes the Microsoft COFF Binary File Dumper (DUMPBIN.EXE), which displays information about Common Object File Format (COFF) binary files.

NMAKE Reference

Describes the Microsoft Program Maintenance Utility (NMAKE.EXE), which is a tool that builds projects based on commands contained in a description file.

MSBuild reference for C++ projects

Article • 08/03/2021

MSBuild is the native build system for all projects in Visual Studio, including C++ projects. When you build a project in the Visual Studio integrated development environment (IDE), it invokes the msbuild.exe tool, which in turn consumes the .vcxproj project file, and various .targets and .props files. In general, we strongly recommend using the Visual Studio IDE to set project properties and invoke MSBuild. Manually editing project files can lead to serious problems if not done correctly.

If for some reason you wish to use MSBuild directly from the command line, see Use MSBuild from the command line. For more information about MSBuild in general, see MSBuild in the Visual Studio documentation.

In this section

MSBuild internals for C++ projects
Information about how properties and targets are stored and consumed.

Common macros for build commands and properties
Describes macros (compile-time constants) that can be used to define properties such as paths and product versions.

File types created for C++ projects
Describes the various kinds of files that Visual Studio creates for different project types.

Visual Studio C++ project templates
Describes the MSBuild-based project types that are available for C++.

C++ new item templates
Describes source files and other items you can add to a Visual Studio project.

Precompiled header files How to use precompiled header files and how to create your own custom precompiled code to speed up build times.

Visual Studio project property reference
Reference documentation for project properties that are set in the Visual Studio IDE.

See also

C/C++ Building Reference

MSBuild internals for C++ projects

Article • 02/14/2022

When you set project properties in the IDE and then save the project, Visual Studio writes the project settings to your project file. The project file contains settings that are unique to your project. However, it doesn't contain all the settings required to build your project. The project file contains `Import` elements that include a network of additional *support files*. The support files contain the remaining properties, targets, and settings required to build the project.

Most targets and properties in the support files exist solely to implement the build system. This article discusses useful targets and properties you can specify on the MSBuild command line. To discover more targets and properties, explore the files in the support file directories.

Support File Directories

By default, the primary Visual Studio support files are located in the following directories. This information is version-specific.

Visual Studio 2022 and 2019

- *`%VSINSTALLDIR%MSBuild\Microsoft\VC\<version>\`*

 Contains the primary target files (*`.targets`*) and property files (*`.props`*) that are used by the targets. By default, the `$(VCTargetsPath)` macro references this directory. The *`<version>`* placeholder refers to the Visual Studio version: v170 for Visual Studio 2022, v160 for Visual Studio 2019, or v150 for Visual Studio 2017.

- *`%VSINSTALLDIR%MSBuild\Microsoft\VC\<version>\Platforms\<platform>\`*

 Contains platform-specific target and property files that override targets and properties in its parent directory. This directory also contains a DLL that defines the tasks that are used by the targets in this directory. The *`<platform>`* placeholder represents the ARM, ARM64, Win32, or x64 subdirectory.

- *`%VSINSTALLDIR%MSBuild\Microsoft\VC\<version>\Platforms\<platform>\PlatformToolsets\<toolset>\`*

 Contains the directories that enable the build to generate C++ applications by using the specified *`<toolset>`*. The *`<platform>`* placeholder represents the ARM,

ARM64, Win32, or x64 subdirectory. The `<toolset>` placeholder represents the toolset subdirectory.

Visual Studio 2017

- `%VSINSTALLDIR%Common7\IDE\VC\VCTargets\`

 Contains the primary target files (`.targets`) and property files (`.props`) that are used by the targets. By default, the `$(VCTargetsPath)` macro references this directory.

- `%VSINSTALLDIR%Common7\IDE\VC\VCTargets\Platforms\<platform>\`

 Contains platform-specific target and property files that override targets and properties in its parent directory. This directory also contains a DLL that defines the tasks that are used by the targets in this directory. The `<platform>` placeholder represents the ARM, ARM64, Win32, or x64 subdirectory.

- `%VSINSTALLDIR%Common7\IDE\VC\VCTargets\Platforms\<platform>\PlatformToolsets\<toolset>\`

 Contains the directories that enable the build to generate C++ applications by using the specified `<toolset>`. The `<platform>` placeholder represents the ARM, Win32, or x64 subdirectory. The `<toolset>` placeholder represents the toolset subdirectory.

Visual Studio 2015 and earlier

- `<drive>:\Program Files[ (x86)]\MSBuild\Microsoft.Cpp\v4.0\<version>\`

 Contains the primary target files (`.targets`) and property files (`.props`) that are used by the targets. By default, the $(VCTargetsPath) macro references this directory.

- `<drive>:\Program Files[ (x86)]\MSBuild\Microsoft.Cpp\v4.0\<version>\Platforms\<platform>\`

 Contains platform-specific target and property files that override targets and properties in its parent directory. This directory also contains a DLL that defines the tasks that are used by the targets in this directory. The `<platform>` placeholder represents the ARM, Win32, or x64 subdirectory.

- *<drive>:\Program Files[(x86)]\MSBuild\Microsoft.Cpp\v4.0\<version>\Platforms\<platform>\PlatformToolsets\<toolset>*

 Contains the directories that enable the build to generate C++ applications by using the specified *<toolset>*. The *<version>* placeholder is V110 for Visual Studio 2012, V120 for Visual Studio 2013, and V140 for Visual Studio 2015. The *<platform>* placeholder represents the ARM, Win32, or x64 subdirectory. The *<toolset>* placeholder represents the toolset subdirectory. For example, it's v140 for building Windows apps by using the Visual Studio 2015 toolset. Or, v120_xp to build for Windows XP using the Visual Studio 2013 toolset.

- *<drive>:\Program Files[(x86)]\MSBuild\Microsoft.Cpp\v4.0\Platforms\<platform>\PlatformToolsets\<toolset>*

 The paths that enable the build to generate either Visual Studio 2008 or Visual Studio 2010 applications don't include the *<version>*. In those versions, the *<platform>* placeholder represents the Itanium, Win32, or x64 subdirectory. The *<toolset>* placeholder represents the v90 or v100 toolset subdirectory.

Support Files

The support file directories contain files with these extensions:

Extension	Description
.targets	Contains `Target` XML elements that specify the tasks that are executed by the target. May also contain `PropertyGroup`, `ItemGroup`, `ItemDefinitionGroup`, and user-defined `Item` elements that are used to assign files and command-line options to task parameters. For more information, see Target Element (MSBuild).
.props	Contains `Property Group` and user-defined `Property` XML elements that specify file and parameter settings that are used during a build. May also contain `ItemDefinitionGroup` and user-defined `Item` XML elements that specify additional settings. Items defined in an item definition group resemble properties, but can't be accessed from the command line. Visual Studio project files frequently use items instead of properties to represent settings. For more information, see ItemGroup Element (MSBuild), ItemDefinitionGroup Element (MSBuild), and Item Element (MSBuild).

Extension	Description
`.xml`	Contains XML elements that declare and initialize IDE user interface elements. For example, property sheets, property pages, textbox controls, and listbox controls. The `.xml` files directly support the IDE, not MSBuild. However, the values of IDE properties are assigned to build properties and items. Most `.xml` files are in a locale-specific subdirectory. For example, files for the English-US region are in `$(VCTargetsPath)\1033\`.

User targets and properties

To use MSBuild effectively, it helps to know which properties and targets are useful and relevant. Most properties and targets help implement the Visual Studio build system, and aren't relevant to the user. This section describes user-oriented properties and targets worth knowing about.

`PlatformToolset` property

The `PlatformToolset` property determines which MSVC toolset is used in the build. By default, the current toolset is used. When this property is set, its value gets concatenated with literal strings to form the path. It's the directory that contains the property and target files required to build a project for a particular platform. The platform toolset must be installed to build by using that platform toolset version.

For example, set the `PlatformToolset` property to `v140` to use Visual Studio 2015 tools and libraries to build your application:

```
msbuild myProject.vcxproj /p:PlatformToolset=v140
```

`PreferredToolArchitecture` property

The `PreferredToolArchitecture` property determines whether the 32-bit or 64-bit compiler and tools are used in the build. This property doesn't affect the output platform architecture or configuration. By default, MSBuild uses the x86 version of the compiler and tools if this property isn't set.

For example, set the `PreferredToolArchitecture` property to `x64` to use the 64-bit compiler and tools to build your application:

```
msbuild myProject.vcxproj /p:PreferredToolArchitecture=x64
```

`UseEnv` property

By default, the platform-specific settings for the current project override the `PATH`, `INCLUDE`, `LIB`, `LIBPATH`, `CONFIGURATION`, and `PLATFORM` environment variables. Set the `UseEnv` property to **`true`** to guarantee that the environment variables don't get overridden.

```
msbuild myProject.vcxproj /p:UseEnv=true
```

Targets

There are hundreds of targets in the Visual Studio support files. However, most are system-oriented targets that the user can ignore. Most system targets are prefixed by an underscore (`_`), or have a name that starts with `PrepareFor`, `Compute`, `Before`, `After`, `Pre`, or `Post`.

The following table lists several useful user-oriented targets.

Target	Description
`BscMake`	Executes the Microsoft Browse Information Maintenance Utility tool, *`bscmake.exe`*.
`Build`	Builds the project. This target is the default for a project.
`ClCompile`	Executes the MSVC compiler tool, *`cl.exe`*.
`Clean`	Deletes temporary and intermediate build files.
`Lib`	Executes the Microsoft 32-Bit Library Manager tool, *`lib.exe`*.
`Link`	Executes the MSVC linker tool, *`link.exe`*.
`ManifestResourceCompile`	Extracts a list of resources from a manifest and then executes the Microsoft Windows Resource Compiler tool, *`rc.exe`*.
`Midl`	Executes the Microsoft Interface Definition Language (MIDL) compiler tool, *`midl.exe`*.
`Rebuild`	Cleans and then builds your project.
`ResourceCompile`	Executes the Microsoft Windows Resource Compiler tool, *`rc.exe`*.
`XdcMake`	Executes the XML Documentation tool, *`xdcmake.exe`*.

Target	Description
`Xsd`	Executes the XML Schema Definition tool, *`xsd.exe`*. *See note.*

> **Note**
>
> In Visual Studio 2017 and later, C++ project support for *`.xsd`* files is deprecated. You can still use `Microsoft.VisualC.CppCodeProvider` by adding *`CppCodeProvider.dll`* manually to the GAC.

See also

MSBuild task reference
BscMake task
CL task
CPPClean task
LIB task
Link task
MIDL task
MT task
RC task
SetEnv task
VCMessage task
XDCMake task

Common macros for MSBuild commands and properties

Article • 01/12/2024

Depending on your installation options, Visual Studio can make hundreds of macros available to you in an MSBuild-based `.vcxproj` Visual Studio project. The macros correspond to the MSBuild properties that are set by default, or in `.props` or `.targets` files, or in your project settings. You can use these macros anywhere in a project's **Property Pages** dialog box where strings are accepted. These macros aren't case-sensitive.

View the current properties and macros

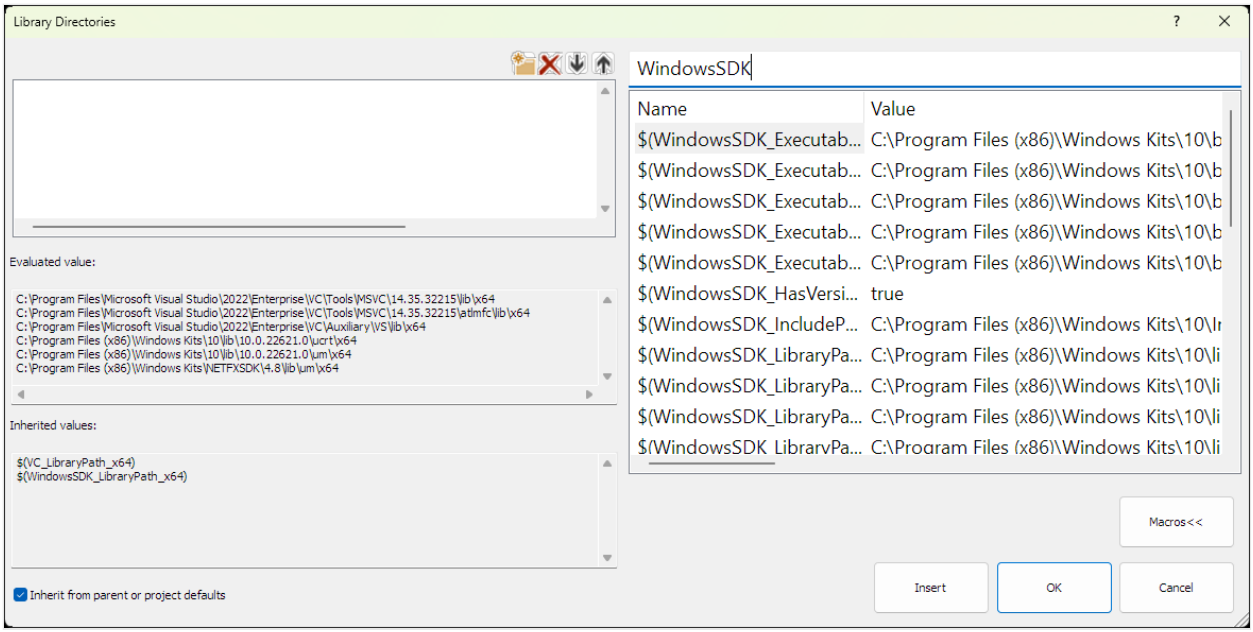
To display all of the currently available macros, open the project property pages from the main menu by selecting **Project** > **Properties**. In the **Property Pages** dialog, choose an entry that has a macro in it. You can recognize a macro by the dollar sign and parenthesis that surround its name.

For example, in the left pane, select **Configuration Properties** > **VC++ Directories**, and then in the right pane, select **Include directories**. The value for **Include directories** is `$(VC_IncludePath);$(WindowsSDK_IncludePath);`.

The dollar sign and parenthesis surrounding these two values indicates that they're macros. The expansion of those two macros sets the include directories to search.

Select **Include Directories** and a dropdown appears at the end of the row. Select the dropdown button, then select **Edit**. In the **Include Directories** dialog box that appears, select the **Macros>>** button.

That expands the dialog to show the current set of properties and macros visible to Visual Studio, along with the current value for each. For more information, see the **Specifying User-Defined Values** section of C++ project property page reference.



List of common macros

This table describes a commonly used subset of the available macros; there are many more not listed here. Go to the **Macros** dialog to see all of the properties and their current values in your project. For details on how MSBuild property definitions are created and used as macros in `.props`, `.targets`, and `.vcxproj` files, see MSBuild Properties.

Expand table

Macro	Description
`$(Configuration)`	The name of the current project configuration, for example, "Debug".
`$(DevEnvDir)`	The installation directory of Visual Studio (defined as drive + path); includes the trailing backslash (\).
`$(FrameworkDir)`	The directory into which the .NET Framework was installed.
`$(FrameworkSDKDir)`	The directory into which you installed the .NET Framework. The .NET Framework may have been installed as part of Visual Studio or separately.
`$(FrameworkVersion)`	The version of the .NET Framework used by Visual Studio. Combined with `$(FrameworkDir)`, the full path to the version of the .NET Framework use by Visual Studio.
`$(FxCopDir)`	The path to the `fxcop.cmd` file. The `fxcop.cmd` file isn't installed in all Visual Studio editions.
`$(IntDir)`	Path to the directory specified for intermediate files. If it's a relative path, intermediate files go to this path appended to the project directory. This path should have a trailing backslash (\). It resolves to the value for the

Macro	Description
	Intermediate Directory property. Don't use `$(OutDir)` to define this property.
`$(OutDir)`	Path to the output file directory. If it's a relative path, output files go to this path appended to the project directory. This path should have a trailing backslash (\). It resolves to the value for the **Output Directory** property. Don't use `$(IntDir)` to define this property.
`$(Platform)`	The name of current project platform, for example, "Win32".
`$(PlatformShortName)`	The short name of current architecture, for example, "x86" or "x64".
`$(ProjectDir)`	The directory of the project (defined as drive + path); includes the trailing backslash (\).
`$(ProjectExt)`	The file extension of the project. It includes the '.' before the file extension.
`$(ProjectFileName)`	The file name of the project (defined as base name + file extension).
`$(ProjectName)`	The base name of the project.
`$(ProjectPath)`	The absolute path name of the project (defined as drive + path + base name + file extension).
`$(PublishDir)`	The output location for the publish target; includes the trailing backslash (\). Defaults to the `$(OutDir)app.publish\` folder.
`$(RemoteMachine)`	Set to the value of the **Remote Machine** property on the Debug property page. For more information, see Changing Project Settings for a C/C++ Debug Configuration.
`$(RootNameSpace)`	The namespace, if any, containing the application.
`$(SolutionDir)`	The directory of the solution (defined as drive + path); includes the trailing backslash (\). Defined only when building a solution in the IDE.
`$(SolutionExt)`	The file extension of the solution. It includes the '.' before the file extension. Defined only when building a solution in the IDE.
`$(SolutionFileName)`	The file name of the solution (defined as base name + file extension). Defined only when building a solution in the IDE.
`$(SolutionName)`	The base name of the solution. Defined only when building a solution in the IDE.
`$(SolutionPath)`	The absolute path name of the solution (defined as drive + path + base name + file extension). Defined only when building a solution in the IDE.
`$(TargetDir)`	The directory of the primary output file for the build (defined as drive +

Macro	Description
	path); includes the trailing backslash (\).
`$(TargetExt)`	The file extension of the primary output file for the build. It includes the '.' before the file extension.
`$(TargetFileName)`	The file name of the primary output file for the build (defined as base name + file extension).
`$(TargetName)`	The base name of the primary output file for the build.
`$(TargetPath)`	The absolute path name of the primary output file for the build (defined as drive + path + base name + file extension).
`$(VCInstallDir)`	The directory that contains the C++ content of your Visual Studio installation. This property contains the version of the targeted Microsoft C++ (MSVC) toolset, which might be different that the host Visual Studio. For example, when building with `$(PlatformToolset) = v140`, `$(VCInstallDir)` contains the path to the Visual Studio 2015 installation.
`$(VSInstallDir)`	The directory into which you installed Visual Studio. This property contains the version of the targeted Visual Studio toolset, which might be different that the host Visual Studio. For example, when building with `$(PlatformToolset) = v110`, `$(VSInstallDir)` contains the path to the Visual Studio 2012 installation.
`$(WebDeployPath)`	The relative path from the web deployment root to where the project outputs belong.
`$(WebDeployRoot)`	The absolute path to the location of `<localhost>`. For example, `c:\inetpub\wwwroot`.

Obsolete macros

The build system for C++ was changed significantly between Visual Studio 2008 and Visual Studio 2010. Many macros used in earlier project types changed to new ones. These macros are no longer used or are replaced by one or more equivalent properties or item metadata macro (`%(item-name)`) values. The migration tool can update macros marked "migrated". If a project containing the macro is migrated from Visual Studio 2008 or earlier to Visual Studio 2010, Visual Studio converts the macro to the equivalent current macro. Later versions of Visual Studio can't convert projects from Visual Studio 2008 and earlier to the new project type. You must convert these projects in two steps; first convert them to Visual Studio 2010, and then convert the result to your newer version of Visual Studio. For more information, see Overview of potential upgrade issues.

Expand table

Macro	Description
`$(InputDir)`	(Migrated.) The directory of the input file (defined as drive + path); includes the trailing backslash (\). If the project is the input, then this macro is equivalent to `$(ProjectDir)`.
`$(InputExt)`	(Migrated.) The file extension of the input file. It includes the '.' before the file extension. If the project is the input, then this macro is equivalent to `$(ProjectExt)`. For source files, it's equivalent to `%(Extension)`.
`$(InputFileName)`	(Migrated.) The file name of the input file (defined as base name + file extension). If the project is the input, then this macro is equivalent to `$(ProjectFileName)`. For source files, it's equivalent to `%(Identity)`.
`$(InputName)`	(Migrated.) The base name of the input file. If the project is the input, then this macro is equivalent to `$(ProjectName)`. For source files, it's equivalent to `%(Filename)`.
`$(InputPath)`	(Migrated.) The absolute path name of the input file (defined as drive + path + base name + file extension). If the project is the input, then this macro is equivalent to `$(ProjectPath)`. For source files, it's equivalent to `%(FullPath)`.
`$(ParentName)`	Name of the item containing this project item. This macro is the parent folder name, or project name.
`$(SafeInputName)`	The name of the file as a valid class name, minus file extension. This property doesn't have an exact equivalent.
`$(SafeParentName)`	The name of the immediate parent in valid name format. For example, a form is the parent of a *`.resx`* file. This property doesn't have an exact equivalent.
`$(SafeRootNamespace)`	The namespace name where the project wizards should add code. This namespace name only contains characters that would be permitted in a valid C++ identifier. This property doesn't have an exact equivalent.

See also

Visual Studio Projects - C++
Visual C++ porting and upgrading guide
Overview of potential upgrade issues
MSBuild well-known item metadata

File Types Created for Visual Studio C++ Projects

Article • 08/03/2021

Many types of files are associated with Visual Studio projects for classic desktop applications. The actual files included in your project depend on the project type and the options you select when using a wizard.

- Project and Solution Files
- CLR Projects
- ATL Program or Control Source and Header Files
- MFC Program or Control Source and Header Files
- Precompiled Header Files
- Resource Files
- Help Files (WinHelp)
- Hint Files

When you create a Visual Studio project, you might create it in a new solution, or you might add a project to an existing solution. Non-trivial applications are commonly developed with multiple projects in a solution.

Projects usually produce either an EXE or a DLL. Projects can be dependent on each other; during the build process, the Visual Studio environment checks dependencies both within and between projects. Each project usually has core source code. Depending on the kind of project, it may have many other files containing various aspects of the project. The contents of these files are indicated by the file extension. The Visual Studio development environment uses the file extensions to determine how to handle the file contents during a build.

The following table shows common files in a Visual Studio project, and identifies them with their file extension.

File extension	Type	Contents
.asmx	Source	Deployment file.

File extension	Type	Contents
.asp	Source	Active Server Page file.
.atp	Project	Application template project file.
.bmp, .dib, .gif, .jpg, .jpe, .png	Resource	General image files.
.bsc	Compiling	The browser code file.
.cpp, .c	Source	Main source code files for your application.
.cur	Resource	Cursor bitmap graphic file.
.dbp	Project	Database project file.
.disco	Source	The dynamic discovery document file. Handles XML Web service discovery.
.exe, .dll	Project	Executable or dynamic-link library files.
.h	Source	A header (include) file.
.htm, .html, .xsp, .asp, .htc, .hta, .xml	Resource	Common Web files.
.HxC	Project	Help project file.
.ico	Resource	Icon bitmap graphic file.
.idb	Compiling	The state file, containing dependency information between source files and class definitions. It can be used by the compiler during incremental compilation. Use the /Fd compiler option to specify the name of the .idb file.
.idl	Compiling	An interface definition language file. For more information, see Interface Definition (IDL) File in the Windows SDK.
.ilk	Linking	Incremental link file. For more information, see /INCREMENTAL.
.map	Linking	A text file containing linker information. Use the /Fm compiler option to name the map file. For more information, see /MAP.
.mfcribbon-ms	Resource	A resource file that contains the XML code that defines the MFC buttons, controls, and attributes in the ribbon. For more information, see Ribbon Designer.
.obj, .o		Object files, compiled but not linked.

File extension	Type	Contents
.pch	Debug	Precompiled header file.
.rc, .rc2	Resource	Resource script files to generate resources.
.sbr	Compiling	Source browser intermediate file. The input file for BSCMAKE.
.sln	Solution	The solution file.
.suo	Solution	The solution options file.
.txt	Resource	A text file, usually the "readme" file.
.vap	Project	A Visual Studio Analyzer project file.
.vbg	Solution	A compatible project group file.
.vbp, .vip, .vbproj	Project	The Visual Basic project file.
.vcxitems	Project	Shared Items project for sharing code files between multiple C++ projects. For more information, see Project and Solution Files.
.vcxproj	Project	The Visual Studio project file. For more information, see Project and Solution Files.
.vcxproj.filters	Project	Used when you use Solution Explorer to add a file to a project. The filters file defines where in the Solution Explorer tree view to add the file, based on its file name extension.
.vdproj	Project	The Visual Studio deployment project file.
.vmx	Project	The macro project file.
.vup	Project	The utility project file.

For information on other files associated with Visual Studio, see File Types and File Extensions in Visual Studio .NET.

Project files are organized into folders in Solution Explorer. Visual Studio creates a folder for source files, header files, and resource files, but you can reorganize these folders or create new ones. You can use folders to organize explicitly logical clusters of files within the hierarchy of a project. For example, you could create folders to contain all your user interface source files. Or, folders for specifications, documentation, or test suites. All file folder names should be unique.

When you add an item to a project, you add the item to all configurations for that project. The item is added whether it's buildable or not. For example, if you have a

project named MyProject, adding an item adds it to both the Debug and Release project configurations.

See also

Creating and Managing Visual Studio C++ Projects
Visual Studio C++ Project Types

Project and Solution Files

Article • 08/03/2021

The following files are created when you create a project in Visual Studio. They are used to manage project files in the solution.

Filename	Directory location	Solution Explorer location	Description
Solname.sln	*Projname*	Not displayed in Solution Explorer	The *solution* file. It organizes all elements of a project or multiple projects into one solution.
Projname.suo	*Projname*	Not displayed in Solution Explorer	The *solution options* file. It stores your customizations for the solution so that every time you open a project or file in the solution, it has the appearance and behavior you want.
Projname.vcxproj	*Projname*	Not displayed in Solution Explorer	The *project* file. It stores information specific to each project. (In earlier versions, this file was named *Projname*.vcproj or *Projname*.dsp.) For an example of a C++ project file (.vcxproj), see Project Files.
Projname.vcxitems	*Projname*	Not displayed in Solution Explorer	The *Shared Items project* file. This project isn't built. Instead, the project can be referenced by another C++ project, and its files will become part of the referencing project's build process. This can be used to share common code with cross-platform C++ projects.
Projname.sdf	*Projname*	Not displayed in Solution Explorer	The *browsing database* file. It supports browsing and navigation features such as **Goto Definition**, **Find All References**, and **Class View**. It is generated by parsing the header files.
Projname.vcxproj.filters	*Projname*	Not displayed in Solution Explorer	The *filters* file. It specifies where to put a file that is added to the solution. For example, a .h file is put in the **Header Files** node.

Filename	Directory location	Solution Explorer location	Description
Projname.vcxproj.user	*Projname*	Not displayed in Solution Explorer	The *migration user* file. After a project is migrated from Visual Studio 2008, this file contains information that was converted from any .vsprops file.
Projname.idl	*Projname*	Source	(Project-specific) Contains the Interface Description Language (IDL) source code for a control type library. This file is used by Visual C++ to generate a type library. The generated library exposes the interface of the control to other Automation clients. For more information, see Interface Definition (IDL) File in the Windows SDK.
Readme.txt	*Projname*	Project	The *read me* file. It is generated by the application wizard and describes the files in a project.

See also

File Types Created for Visual Studio C++ projects

C++ project templates

Article • 08/03/2021

Visual Studio project templates generate source code files, compiler options, menus, toolbars, icons, references, and `#include` statements that are appropriate for the kind of project you want to create. Visual Studio includes several kinds of C++ project templates and provides wizards for many of them so that you can customize your projects as you create them. Immediately after you create a project, you can build it and run the application; it's good practice to build intermittently as you develop your application.

> **Note**
>
> You can create a C-language project by using C++ project templates. In the generated project, locate files that have a .cpp file name extension and change it to .c. Then, on the **Project Properties** page for the project (not for the solution), expand **Configuration Properties**, **C/C++** and select **Advanced**. Change the **Compile As** setting to **Compile as C Code (/TC)**.

Project templates

The project templates included in Visual Studio depend on the product version and the workloads you've installed. If you've installed the Desktop development with C++ workload, Visual Studio has these C++ project templates.

Windows Desktop

Project template	Description
Windows Console Application	A project for creating a Windows console application.
Windows Desktop Application	A project for creating a Windows desktop (Win32) application.
Dynamic-Link Library	A project for creating a dynamic-link library (DLL).
Static Library	A project for creating a static library (LIB).
Windows Desktop Wizard	A wizard for creating Windows desktop applications and libraries with additional options.

General

Project template	Description
Empty Project	An empty project for creating an application, library, or DLL. You must add any code or resources required.
Makefile Project	A project that wraps a Windows makefile in a Visual Studio project. (To open a makefile as-is in Visual Studio, use Open Folder.
Shared Items Project	A project used for sharing code files or resource files between multiple projects. This project type does not produce an executable file.

ATL

Project template	Description
ATL Project	A project that uses the Active Template Library.

Test

Project template	Description
Native Unit Test Project	A project that contains native C++ unit tests.

MFC

If you add the MFC and ATL support component to your Visual Studio installation, these project templates are added to Visual Studio.

Project template	Description
MFC Application	A project for creating an application that uses the Microsoft Foundation Class (MFC) Library.
MFC ActiveX Control	A project for creating an ActiveX control that uses the MFC library.
MFC DLL	A project for creating a dynamic-link library that uses the MFC library.

Windows Universal Apps

If you add the C++ Windows Universal Platform tools component to your Visual Studio installation, these project templates are added to Visual Studio.

For an overview of Windows Universal apps in C++, see Universal Windows Apps (C++).

Project template	Description
Blank App	A project for a single-page Universal Windows Platform (UWP) app that has no predefined controls or layout.
DirectX 11 App	A project for a Universal Windows Platform app that uses DirectX 11.
DirectX 12 App	A project for a Universal Windows Platform app that uses DirectX 12.
DirectX 11 and XAML App	A project for a Universal Windows Platform app that uses DirectX 11 and XAML.
Unit Test App	A project to create a unit test app for Universal Windows Platform (UWP) apps.
DLL	A project for a native dynamic-link library (DLL) that can be used by a Universal Windows Platform app or runtime component.
Static Library	A project for a native static link library (LIB) that can be used by a Universal Windows Platform app or runtime component.
Windows Runtime Component	A project for a Windows Runtime component that can be used by a Universal Windows Platform app, regardless of the programming language in which the app is written.
Windows Application Packaging Project	A project that creates a UWP package that enables a desktop application to be side-loaded or distributed via the Microsoft Store.

TODO Comments

Many of the files generated by a project template contain TODO comments to help you identify where you can provide your own source code. For more information about how to add code, see Adding Functionality with Code Wizards and Working with Resource Files.

Using Visual C++ Add New Item Templates

Article • 08/03/2021

You can easily add items that are common to Visual Studio projects by using the **Add New Item** command. When you use the **Add New Item** command, the `Add New Item` dialog box appears with a list of item templates, which add the appropriate files to your project.

The following table is an alphabetical list of Visual Studio Add New Item templates.

Template	Description
Assembly Resource File (.resx)	Creates a file containing CLR resources.
Bitmap File (.bmp)	Creates a Win32 bitmap file.
C++ File (.cpp)	Creates a C++ source file.
Class Diagram (.cd)	Creates an empty class diagram.
Code Analysis Rule Set (.ruleset)	Creates a settings file for configuring Code Analysis.
Configuration File (app.config)	Creates an empty configuration file.
Component Class	Adds a Component Class using CLR features.
Cursor File (.cur)	Creates a Win32 cursor file.
Discovery File, Static (.disco)	Creates a static discovery file, which is an XML document that contains links to other resources that describe the XML Web service, enables programmatic discovery of an XML Web service.
Frameset (.htm)	Adds an HTML file that hosts multiple HTML pages.

Template	Description
Header File (.h)	Creates a C++ header file.
HTML Page (.htm)	Creates a blank HTML file.
Icon File (.ico)	Creates a Win32 icon file.
Installer Class	Adds a class that inherits from the Installer using CLR features.
MIDL File (.idl)	Creates an Interface Definition Language file.
Module-Definition File (.def)	Creates a DLL export definition file.
Property Sheet (.props)	Creates a property sheet file.
Registration Script (.rgs)	Creates an ATL registration script file.
Report (.rdlc)	Creates a report file.
Resource File (.rc)	Creates a Win32 resource file.
Resource Template File (.rct)	Creates a resource template file.
Ribbon (.mfcribbon-ms)	Creates a ribbon file.
Server Response File (.srf)	Creates a server response file that is used with ATL Server.
SQL Script File (.sql)	Creates an SQL script file. **Note:** This template is not a Professional Edition feature.
Style Sheet (.css)	Adds a cascading style sheet used for rich HTML style definitions.
Text File (.txt)	Adds a blank text file.
User Control	Adds a User Control using CLR features.

Template	Description
Windows Form	Adds a Windows Form using CLR features.
XML File (.xml)	Adds a blank XML file.
XML Schema File (.xsd)	Creates a file that is used to define a schema for XML documents.
XSLT File (.xslt)	Creates a file used to transform XML documents.

See also

Adding Functionality with Code Wizards

Project Resource Files (C++)

Article • 08/03/2021

Resources are interface elements that provide information to the user. Bitmaps, icons, toolbars, and cursors are all resources. Some resources can perform an action such as selecting from a menu or entering data in dialog box.

For more information, see Working with Resources.

File name	Directory location	Solution Explorer location	Description
Projname.rc	*Projname*	Source Files	The resource script file for the project. The resource script file contains the following, depending on the type of project, and the support selected for the project (for example, toolbars, dialog boxes, or HTML): - Default menu definition. - Accelerator and string tables. - Default **About** dialog box. - Other dialog boxes. - Icon file (res*Projname*.ico). - Version information. - Bitmaps. - Toolbar. - HTML files. The resource file includes the file Afxres.rc for standard Microsoft Foundation Class resources.
Resource.h	*Projname*	Header Files	The resource header file that includes definitions for the resources used by the project.
Projname.rc2	*Projname*\res	Source Files	The script file containing additional resources used by the project. You can include the .rc2 file under the project's .rc file. An .rc2 file is useful for including resources used by several different projects. Instead of having to create the same resources several times for different projects, you can put them in an .rc2 file and include the .rc2 file into the main .rc file.

File name	Directory location	Solution Explorer location	Description
Projname.def	*Projname*	Source Files	The module definition file for a DLL project. For a control, it provides the name and description of the control, as well as the size of the run-time heap.
Projname.ico	*Projname*\res	Resource Files	The icon file for the project or control. This icon appears when the application is minimized. It is also used in the application's **About** box. By default, MFC provides the MFC icon, and ATL provides the ATL icon.
*Projname*Doc.ico	*Projname*\res	Resource Files	The icon file for an MFC project that includes support for the document/view architecture.
Toolbar.bmp	*Projname*\res	Resource Files	The bitmap file representing the application or control in a toolbar or palette. This bitmap is included in the project's resource file. The initial toolbar and status bar are constructed in the **CMainFrame** class.
ribbon.mfcribbon-ms	*Projname*\res	Resource Files	The resource file that contains the XML code that defines the buttons, controls, and attributes in the ribbon. For more information, see Ribbon Designer (MFC).

See also

File Types Created for Visual Studio C++ projects

Files Created for CLR Projects

Article • 08/03/2021

When you use Visual C++ templates to create your projects, several files are created, depending on which template you use. The following table lists all the files that are created by project templates for .NET Framework projects.

File name	File description
AssemblyInfo.cpp	The file that contains information (that is, attributes, files, resources, types, versioning information, signing information, and so on) for modifying the project's assembly metadata. For more information see Assembly Concepts.
projname.asmx	A text file that references managed classes that encapsulate the functionality of the XML Web service.
projname.cpp	The main source file and entry point into the application that Visual Studio created for you. Identifies the project .dll file and the project namespace. Provide your own code in this file.
projname.vsdisco	An XML deployment file containing links to other resources that describe the XML Web service.
projname.h	The main include file for the project, which contains all declarations, global symbols, and `#include` directives for other header files.
projname.sln	The solution file used within the development environment to organize all elements of your project into a single solution.
projname.suo	The solution options file used within the development environment.
projname.vcxproj	The project file used within the development environment that stores the information specific to this project.
ReadMe.txt	A file describing each file in your project using the actual filenames created by the template.

ATL program or control source and header files

Article • 09/28/2022

The following files are created when you create an ATL project in Visual Studio, depending on the options you select for the project you create. The file names depend on the name you choose for your project, which we'll call *ProjectName*.

All of the files created by the project template are located in the *ProjectName* and *ProjectNamePS* project directories. In Solution Explorer, the *ProjectName* files are located in the **Generated Files**, **Header Files**, **Resource Files**, and **Source Files** folders. The *ProjectNamePS* files are in the **Generated Files** and **Source Files** folders. Not all files listed here are generated for every project type. Files in the **Generated Files** folder are generated automatically by the MIDL compiler; they shouldn't be edited directly.

File name	Description
ProjectName_i.c	The generated source file containing the C++ IID and CLSID definitions and GUID declarations of the items defined in *ProjectName.idl*. Don't edit this file; it's regenerated by MIDL during compilation. Link this file with the server and any clients.
ProjectName_i.h	The generated include file containing the C++ interface declarations and GUID declarations of the items defined in *ProjectName.idl*. Don't edit this file; it's regenerated by MIDL during compilation. Include this file in source files for the server and any clients.
ProjectName.rc	The main program resource file.
ProjectName.rgs	The main program registration file.
ProjectName.cpp	The main program source file. In DLL projects, it contains the implementation of your DLL's exports for an in-process server. In EXE projects, it contains the implementation of `WinMain` for a local server. For a service, this file implements all the service management functions.
ProjectName.def	In DLL projects, the definitions for your DLL's exports.
ProjectName.idl	The IDL source for your project. The MIDL tool processes this file to produce the type library (*.tlb*) and marshaling code.
framework.h	Sets preprocessor macros and includes the ATL header files, the *targetver.h* version support header, and the *Resource.h* resource file header.
dllmain.h	In DLL projects, the header file for the module class.

File name	Description
dllmain.cpp	In DLL projects, the source file for the `DllMain` function.
Resource.h	The header file for the resource file.
targetver.h	Includes *SDKDDKVer.h*. To build your application for a previous Windows platform, include *WinSDKVer.h* and set the `_WIN32_WINNT` macro to the platform you wish to support before including *SDKDDKVer.h*.
pch.cpp	Includes the file *pch.h*.
pch.h	Includes the *framework.h* header file.

See also

File types created for Visual Studio C++ projects
MFC program or control source and header files
Add ATL support to an existing MFC executable or DLL
CLR projects

MFC Program or Control Source and Header Files

Article • 08/03/2021

The following files are created when you create an MFC project in Visual Studio, depending on the options you select for the project you create. For example, your project contains *Projname*dlg.cpp and *Projname*dlg.h files only if you create a dialog-based project or class.

All of these files are located in the *Projname* directory, and in either the Header Files (.h files) folder or Source Files (.cpp files) folder in Solution Explorer.

File name	Description
Projname.h	The main include file for the program or DLL. It contains all global symbols and `#include` directives for other header files. It derives the `CPrjnameApp` class from `CWinApp` and declares an `InitInstance` member function. For a control, the `CPrjnameApp` class is derived from `COleControlModule`.
Projname.cpp	The main program source file. It creates one object of the class `CPrjnameApp`, which is derived from `CWinApp`, and overrides the `InitInstance` member function. For executables, `CPrjnameApp::InitInstance` does several things. It registers document templates, which serve as a connection between documents and views; creates a main frame window; and creates an empty document (or opens a document if one is specified as a command-line argument to the application). For DLLs and ActiveX (formerly OLE) controls, `CProjNameApp::InitInstance` registers the control's object factory with OLE by calling `COleObjectFactory::RegisterAll` and makes a call to `AfxOLEInit`. In addition, the member function `CProjNameApp::ExitInstance` is used to unload the control from memory with a call to **AfxOleTerm**. This file also registers and unregisters the control in the Windows registration database by implementing the `DllRegisterServer` and `DllUnregisterServer` functions.

File name	Description
*Projname*ctrl.h, *Projname*ctrl.cpp	Declare and implement the `CProjnameCtrl` class. `CProjnameCtrl` is derived from `COleControl`, and skeleton implementations of some member functions are defined that initialize, draw, and serialize (load and save) the control. Message, event, and dispatch maps are also defined.
*Projname*dlg.cpp, *Projname*dlg.h	Created if you choose a dialog-based application. The files derive and implement the dialog class, named `CProjnameDlg`, and include skeleton member functions to initialize a dialog and perform dialog data exchange (DDX). Your About dialog class is also placed in these files instead of in *Projname*.cpp.
Dlgproxy.cpp, Dlgproxy.h	In a dialog-based program, the implementation and header file for the project's Automation proxy class for the main dialog. This is only used if you have chosen Automation support.
*Projname*doc.cpp, *Projname*doc.h	Derive and implement the document class, named `CProjnameDoc`, and include skeleton member functions to initialize a document, serialize (save and load) a document, and implement debugging diagnostics.
*Projname*set.h/.cpp	Created if you create a program that supports a database and contains the recordset class.
*Projname*view.cpp, *Projname*view.h	Derive and implement the view class, named `CProjnameView`, which is used to display and print the document data. The `CProjnameView` class is derived from one of the following MFC classes: - CEditView - CFormView - CRecordView - COleDBRecordView - CTreeView - CListView - CRichEditView - CScrollView - CView - CHTMLView - CHTMLEditView The project's view class contains skeleton member functions to draw the view and implement debugging diagnostics. If you have enabled support for printing, then message-map entries are added for print, print setup, and print preview command messages. These entries call the corresponding member functions in the base view class.

File name	Description
*Projname*PropPage.h, *Projname*PropPage.cpp	Declare and implement the `CProjnamePropPage` class. `CProjnamePropPage` is derived from `COlePropertyPage` and a skeleton member function, `DoDataExchange`, is provided to implement data exchange and validation.
IPframe.cpp, IPframe.h	Created if the Mini-Server or Full-Server option is selected in the application wizard's **Automation Options** page (step 3 of 6). The files derive and implement the in-place frame window class, named **CInPlaceFrame**, used when the server is in place activated by a container program.
Mainfrm.cpp, Mainfrm.h	Derive the **CMainFrame** class from either CFrameWnd (for SDI applications) or CMDIFrameWnd (for MDI applications). The **CMainFrame** class handles the creation of toolbar buttons and the status bar, if the corresponding options are selected in the application wizard's **Application Options** page (step 4 of 6). For information on using **CMainFrame**, see The Frame-Window Classes Created by the Application Wizard.
Childfrm.cpp, Childfrm.h	Derive the **CChildFrame** class from CMDIChildWnd. The **CChildFrame** class is used for MDI document frame windows. These files are always created if you select the MDI option.

See also

File Types Created for Visual Studio C++ projects
ATL Program or Control Source and Header Files
CLR Projects

Help Files (HTML Help)

Article • 08/03/2021

The following files are created when you add the HTML Help type of Help support to your application by selecting the **Context-sensitive help** check box and then selecting **HTML Help format** in the Advanced Features page of the MFC Application Wizard.

File name	Directory location	Solution Explorer location	Description
Projname.hhp	*Projname*\hlp	HTML Help files	The help project file. It contains the data needed to compile the help files into an .hxs file or a .chm file.
Projname.hhk	*Projname*\hlp	HTML Help files	Contains an index of the help topics.
Projname.hhc	*Projname*\hlp	HTML Help files	The contents of the help project.
Makehtmlhelp.bat	*Projname*	Source Files	Used by the system to build the Help project when the project is compiled.
Afxcore.htm	*Projname*\hlp	HTML Help Topics	Contains the standard help topics for standard MFC commands and screen objects. Add your own help topics to this file.
Afxprint.htm	*Projname*\hlp	HTML Help Topics	Contains the help topics for the printing commands.
*.jpg; *.gif	*Projname*\hlp\Images	Resource Files	Contain images for the different generated help file topics.

See also

File Types Created for Visual Studio C++ projects

Help Files (WinHelp)

Article • 08/03/2021

The following files are created when you add the WinHelp type of Help support to your application by selecting the **Context-sensitive help** check box and then selecting **WinHelp format** in the Advanced Features page of the MFC Application Wizard.

File name	Directory location	Solution Explorer location	Description
Projname.hpj	*Projname*\hlp	Source Files	The Help project file used by the Help compiler to create your program or control's Help file.
Projname.rtf	*Projname*\hlp	Help Files	Contains template topics that you can edit and information on customizing your .hpj file.
Projname.cnt	*Projname*\hlp	Help Files	Provides the structure for the **Contents** window in Windows Help.
Makehelp.bat	*Projname*	Source Files	Used by the system to build the Help project when the project is compiled.
Print.rtf	*Projname*\hlp	Help Files	Created if your project includes printing support (the default). Describes the printing commands and dialog boxes.
*.bmp	*Projname*\hlp	Resource Files	Contain images for the different generated help file topics.

You can add WinHelp support to an MFC ActiveX Control project by selecting **Generate help files** in the Application Settings tab of the MFC ActiveX Control Wizard. The following files are added to your project when you add Help support to an MFC ActiveX control:

File name	Directory location	Solution Explorer location	Description
Projname.hpj	*Projname*\hlp	Source files	The project file used by the Help compiler to create your program or control's Help file.
Projname.rtf	*Projname*\hlp	Project	Contains template topics that you can edit and information on customizing your .hpj file.
Makehelp.bat	*Projname*	Source Files	Used by the system to build the Help project when the project is compiled.

File name	Directory location	Solution Explorer location	Description
Bullet.bmp	*Projname*	Resource Files	Used by standard Help file topics to represent bulleted lists.

See also

File Types Created for Visual Studio C++ projects

Hint Files

Article • 11/11/2021

A *hint file* contains macros that would otherwise cause regions of code to be skipped by the C++ Browsing Database Parser. When you open a Visual Studio C++ project, the parser analyzes the code in each source file in the project and builds a database with information about every identifier. The IDE uses that information to support code browsing features such as the **Class View** browser and the **Navigation Bar**.

The C++ Browsing Database Parser is a fuzzy parser that can parse large amounts of code in a short amount of time. One reason it's fast is because it skips the content of blocks. For instance, it only records the location and parameters of a function, and ignores its contents. Certain macros can cause issues for the heuristics used to determine the start and end of a block. These issues cause regions of code to be recorded improperly.

These skipped regions can manifest in multiple ways:

- Missing types and functions in **Class View**, **Go To** and **Navigation Bar**
- Incorrect scopes in the **Navigation Bar**
- Suggestions to **Create Declaration/Definition** for functions that are already defined

A hint file contains user-customizable hints, which have the same syntax as C/C++ macro definitions. Visual C++ includes a built-in hint file that is sufficient for most projects. However, you can create your own hint files to improve the parser specifically for your project.

> **ⓘ Important**
>
> If you modify or add a hint file, you need to take additional steps in order for the changes to take effect:
>
> - In versions before Visual Studio 2017 version 15.6: Delete the .sdf file and/or VC.db file in the solution for all changes.
> - In Visual Studio 2017 version 15.6 and later: Close and reopen the solution after adding new hint files.

Scenario

C++

```
#define NOEXCEPT noexcept
void Function() NOEXCEPT
{
}
```

Without a hint file, `Function` doesn't show up in **Class View**, **Go To** or the **Navigation Bar**. After adding a hint file with this macro definition, the parser now understands and replaces the `NOEXCEPT` macro, which allows it to correctly parse the function:

cpp.hint

```
#define NOEXCEPT
```

Disruptive Macros

There are two categories of macros that disrupt the parser:

- Macros that encapsulate keywords that adorn a function

 C++

  ```
  #define NOEXCEPT noexcept
  #define STDMETHODCALLTYPE __stdcall
  ```

 For these types of macros, only the macro name is required in the hint file:

 cpp.hint

  ```
  #define NOEXCEPT
  #define STDMETHODCALLTYPE
  ```

- Macros that contain unbalanced brackets

 C++

  ```
  #define BEGIN {
  ```

 For these types of macros, both the macro name and its contents are required in the hint file:

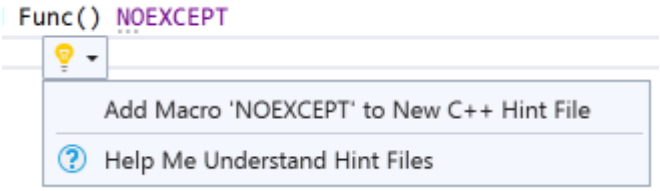
cpp.hint

```
#define BEGIN {
```

Editor Support

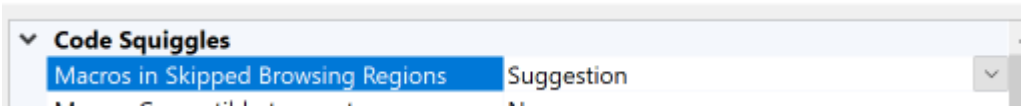
Starting in Visual Studio 2017 version 15.8 there are several features to identify disruptive macros:

- Macros that are inside regions skipped by the parser are highlighted.
- There's a Quick Action to create a hint file that includes the highlighted macro, or if there's an existing hint file, to add the macro to the hint file.



After executing either of the Quick Actions, the parser reparses the files affected by the hint file.

By default, the problem macro is highlighted as a suggestion. The highlight can be changed to something more noticeable, such as a red or green squiggle. Use the **Macros in Skipped Browsing Regions** option in the **Code Squiggles** section under **Tools** > **Options** > **Text Editor** > **C/C++** > **View**.



Display Browsing Database Errors

The **Project** > **Display Browsing Database Errors** menu command displays all the regions that failed to parse in the **Error List**. The command is meant to streamline building the initial hint file. However, the parser can't tell if the cause of the error was a disruptive macro, so you must evaluate each error. Run the **Display Browsing Database Errors** command and navigate to each error to load the affected file in the editor. Once the file is loaded, if any macros are inside the region, they're highlighted. You can invoke the Quick Actions to add them to a hint file. After a hint file update, the error list is updated automatically. Alternatively, if you're modifying the hint file manually you can use the **Rescan Solution** command to trigger an update.

Architecture

Hint files relate to physical directories, not the logical directories shown in **Solution Explorer**. You don't have to add a hint file to your project for the hint file to have an effect. The parsing system uses hint files only when it parses source files.

Every hint file is named **cpp.hint**. Many directories can contain a hint file, but only one hint file can occur in a particular directory.

Your project can be affected by zero or more hint files. If there are no hint files, the parsing system uses error recovery techniques to ignore indecipherable source code. Otherwise, the parsing system uses the following strategy to find and gather hints.

Search Order

The parsing system searches directories for hint files in the following order.

- The directory that contains the installation package for Visual C++ (**vcpackages**). This directory contains a built-in hint file that describes symbols in frequently used system files, such as **windows.h**. Consequently, your project automatically inherits most of the hints that it needs.

- The path from the root directory of a source file to the directory that contains the source file itself. In a typical Visual Studio C++ project, the root directory contains the solution or project file.

 The exception to this rule is if a *stop file* is in the path to the source file. A stop file is any file that is named **cpp.stop**. A stop file provides additional control over the search order. Instead of starting from the root directory, the parsing system searches from the directory that contains the stop file to the directory that contains the source file. In a typical project, you don't need a stop file.

Hint Gathering

A hint file contains zero or more *hints*. A hint is defined or deleted just like a C/C++ macro. That is, the `#define` preprocessor directive creates or redefines a hint, and the `#undef` directive deletes a hint.

The parsing system opens each hint file in the search order described earlier. It accumulates each file's hints into a set of *effective hints*, and then uses the effective hints to interpret the identifiers in your code.

The parsing system uses these rules to accumulate hints:

- If the new hint specifies a name that isn't already defined, the new hint adds the name to the effective hints.
- If the new hint specifies a name that is already defined, the new hint redefines the existing hint.
- If the new hint is an `#undef` directive that specifies an existing effective hint, the new hint deletes the existing hint.

The first rule means that effective hints are inherited from previously opened hint files. The last two rules mean that hints later in the search order can override earlier hints. For example, you can override any previous hints if you create a hint file in the directory that contains a source file.

For a depiction of how hints are gathered, see the Example section.

Syntax

You create and delete hints by using the same syntax as the preprocessor directives to create and delete macros. In fact, the parsing system uses the C/C++ preprocessor to evaluate the hints. For more information about the preprocessor directives, see #define Directive (C/C++) and #undef Directive (C/C++).

The only unusual syntax elements are the `@<`, `@=`, and `@>` replacement strings. These hint-file specific replacement strings are only used in *map* macros. A map is a set of macros that relate data, functions, or events to other data, functions, or event handlers. For example, `MFC` uses maps to create message maps, and `ATL` uses maps to create object maps. The hint-file specific replacement strings mark the starting, intermediate, and ending elements of a map. Only the name of a map macro is significant. Therefore, each replacement string intentionally hides the implementation of the macro.

Hints use this syntax:

Syntax	Meaning

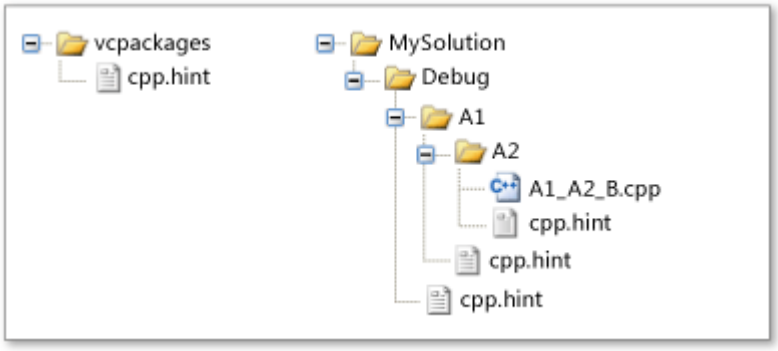
Syntax	Meaning
`#define` *hint-name replacement-string* `#define` *hint-name* `(` *parameter*, ... `)` *replacement-string*	A preprocessor directive that defines a new hint or redefines an existing hint. After the directive, the preprocessor replaces each occurrence of *hint-name* in source code with *replacement-string*. The second syntax form defines a function-like hint. If a function-like hint occurs in source code, the preprocessor first replaces each occurrence of *parameter* in *replacement-string* with the corresponding argument in source code, and then replaces *hint-name* with *replacement-string*.
`@<`	A hint-file specific *replacement-string* that indicates the start of a set of map elements.
`@=`	A hint-file specific *replacement-string* that indicates an intermediate map element. A map can have multiple map elements.
`@>`	A hint-file specific *replacement-string* that indicates the end of a set of map elements.
`#undef` *hint-name*	The preprocessor directive that deletes an existing hint. The name of the hint is provided by the *hint-name* identifier.
`//` *comment*	A single-line comment.
`/*` *comment* `*/`	A multiline comment.

Example

This example shows how hints are accumulated from hint files. Stop files aren't used in this example.

The illustration shows some of the physical directories in a Visual Studio C++ project. There are hint files in the `vcpackages`, `Debug`, `A1`, and `A2` directories.

Hint File Directories



Directories and Hint File Contents

This list shows the directories in this project that contain hint files, and the contents of those hint files. Only some of the many hints in the `vcpackages` directory hint file are listed:

- vcpackages

 cpp.hint

  ```
  // vcpackages (partial list)
  #define _In_
  #define _In_opt_
  #define _In_z_
  #define _In_opt_z_
  #define _In_count_(size)
  ```

- Debug

 cpp.hint

  ```
  // Debug
  #undef _In_
  #define OBRACE {
  #define CBRACE }
  #define RAISE_EXCEPTION(x) throw (x)
  #define START_NAMESPACE namespace MyProject {
  #define END_NAMESPACE }
  ```

- A1

 cpp.hint

  ```
  // A1
  #define START_NAMESPACE namespace A1Namespace {
  ```

- A2

cpp.hint

```
// A2
#undef OBRACE
#undef CBRACE
```

Effective Hints

This table lists the effective hints for the source files in this project:

- Source File: A1_A2_B.cpp
- Effective hints:

cpp.hint

```
// vcpackages (partial list)
#define _In_opt_
#define _In_z_
#define _In_opt_z_
#define _In_count_(size)
// Debug...
#define RAISE_EXCEPTION(x) throw (x)
// A1
#define START_NAMESPACE namespace A1Namespace {
// ...Debug
#define END_NAMESPACE }
```

These notes apply to the preceding list:

- The effective hints are from the `vcpackages`, `Debug`, `A1`, and `A2` directories.
- The **#undef** directive in the `Debug` hint file removed the `#define _In_` hint in the `vcpackages` directory hint file.
- The hint file in the `A1` directory redefines `START_NAMESPACE`.
- The `#undef` hint in the `A2` directory removed the hints for `OBRACE` and `CBRACE` in the `Debug` directory hint file.

See also

File Types Created for Visual Studio C++ projects
#define Directive (C/C++)

#undef Directive (C/C++)
SAL Annotations

Property Page XML rule files

Article • 08/03/2021

The project property pages in the IDE are configured by XML files in the default rules folder. The XML files describe the names of the rules, the categories, and the individual properties, their data type, default values, and how to display them. When you set a property in the IDE, the new value is stored in the project file.

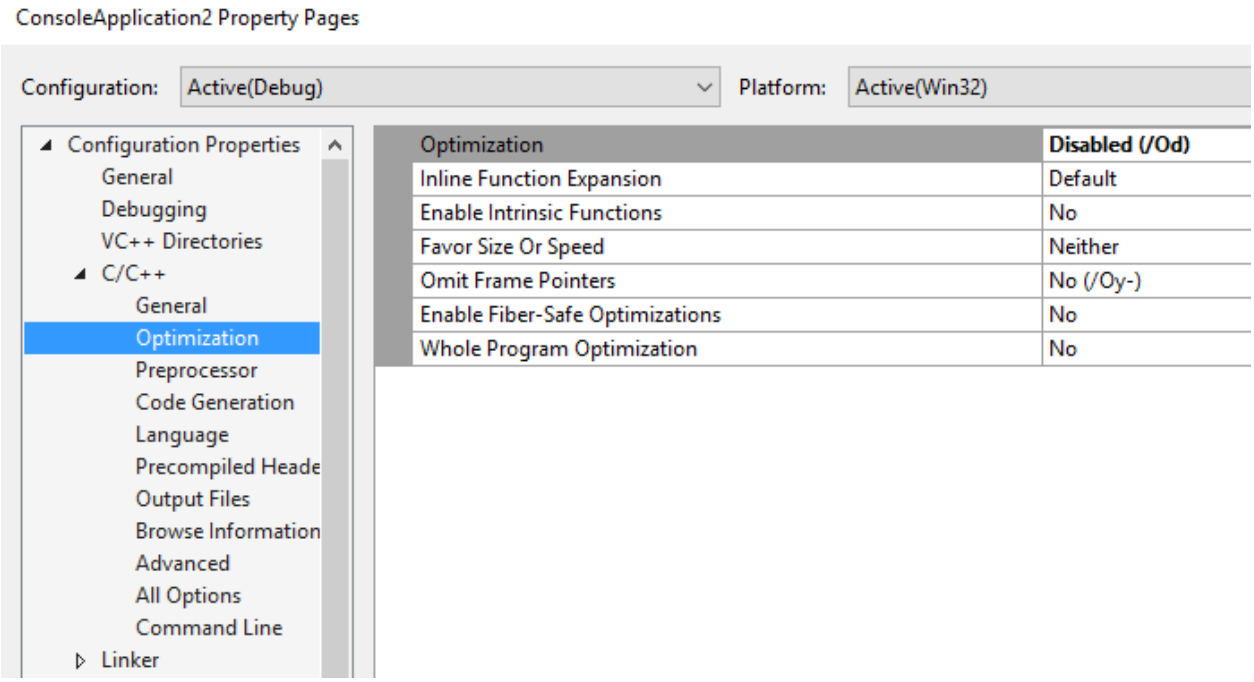
The path to the default rules folder depends on the locale and the version of Visual Studio in use. In a Visual Studio 2019 or later developer command prompt, the rules folder is `%VSINSTALLDIR%MSBuild\Microsoft\VC\<version>\<locale>\`, where the `<version>` value is `v160` in Visual Studio 2019. The `<locale>` is an LCID, for example, `1033` for English. In Visual Studio 2017, the rules folder is `%VSINSTALLDIR%Common7\IDE\VC\VCTargets\<locale>\`. In a Visual Studio 2015 or earlier developer command prompt, the rules folder is `%ProgramFiles(x86)%\MSBuild\Microsoft.Cpp\v4.0\<version>\<locale>\`. You'll use a different path for each edition of Visual Studio that's installed, and for each language. For example, the default rules folder path for Visual Studio 2019 Community edition in English could be `C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\MSBuild\Microsoft\VC\v160\1033\`.

You only need to understand the internal workings of these files and the Visual Studio IDE in a couple of scenarios:

- You want to create a custom property page, or
- You want to customize your project properties without use of the Visual Studio IDE.

Contents of rule files

First, let's open the property pages for a project. Right-click on the project node in **Solution Explorer** and choose **Properties**:



Each node under **Configuration Properties** is called a *rule*. A rule sometimes represents a single tool like the compiler. In general, the term refers to something that has properties, that executes and that may produce some output. Each rule is populated from an XML file in the default rules folder. For example, the C/C++ rule that's shown here is populated by `cl.xml`.

Each rule has a set of properties, which are organized into *categories*. Each sub-node under a rule represents a category. For example, the **Optimization** node under **C/C++** contains all the optimization-related properties of the compiler tool. The properties and their values get rendered in a grid format on the right pane.

You can open `cl.xml` in notepad or any XML editor. You'll see a root node called `Rule`. It defines the same list of properties that get displayed in the UI, along with additional metadata.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--Copyright, Microsoft Corporation, All rights reserved.-->
<Rule Name="CL" PageTemplate="tool" DisplayName="C/C++" SwitchPrefix="/"
Order="10" xmlns="http://schemas.microsoft.com/build/2009/properties"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:sys="clr-
namespace:System;assembly=mscorlib">
  <Rule.Categories>
    <Category Name="General" DisplayName="General" />
    <Category Name="Optimization" DisplayName="Optimization" />
    <Category Name="Preprocessor" DisplayName="Preprocessor" />
    <Category Name="Code Generation" DisplayName="Code Generation" />
    <Category Name="Language" DisplayName="Language" />
    <Category Name="Precompiled Headers" DisplayName="Precompiled Headers"
/>
```

```
    <Category Name="Output Files" DisplayName="Output Files" />
    <Category Name="Browse Information" DisplayName="Browse Information" />
    <Category Name="Advanced" DisplayName="Advanced" />
    <Category Name="All Options" DisplayName="All Options" Subtype="Search"
/>
    <Category Name="Command Line" DisplayName="Command Line"
Subtype="CommandLine" />
  </Rule.Categories>
  <!-- . . . -->
</Rule>
```

There's one XML file for every node under **Configuration Properties** in the property pages UI. You can add or remove rules in the UI: it's done by including or removing locations to corresponding XML files in the project. For example, it's how `Microsoft.CppBuild.targets` (found one level higher than the 1033 folder) includes `cl.xml`:

XML

```
<PropertyPageSchema Condition="'$(ConfigurationType)' != 'Utility'"
Include="$(VCTargetsPath)$(LangID)\cl.xml"/>
```

If you strip `cl.xml` of all data, you have this basic framework:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<Rule>
  <Rule.DataSource />
  <Rule.Categories>
    <Category />
    <!-- . . . -->
  </Rule.Categories>
  <BoolProperty />
  <EnumProperty />
  <IntProperty />
  <StringProperty />
  <StringListProperty />
</Rule>
```

The next section describes each major element and some of the metadata that you can attach.

Rule attributes

A `<Rule>` element is the root node in the XML file. It can have many attributes:

XML

```xml
<Rule Name="CL" PageTemplate="tool" SwitchPrefix="/" Order="10"
          xmlns="http://schemas.microsoft.com/build/2009/properties"
          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
          xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <Rule.DisplayName>
    <sys:String>C/C++</sys:String>
  </Rule.DisplayName>
```

- **`Name`**: The Name attribute is an ID for the `Rule`. It needs to be unique among all the property page XML files for a project.
- **`PageTemplate`**: The value of this attribute is used by the UI to choose from a collection of UI templates. The "tool" template renders the properties in a standard grid format. Other in-built values for this attribute are "debugger" and "generic". See the Debugging node and General node, respectively, to see the UI format resulting from specifying these values. The UI for "debugger" page template uses a drop-down box to switch between the properties of different debuggers. The "generic" template displays different property categories all in one page, as opposed to having multiple category sub-nodes under the `Rule` node. This attribute is just a suggestion to the UI. The XML file is designed to be UI independent. A different UI might use this attribute for different purposes.
- **`SwitchPrefix`**: The prefix used in the command line for the switches. A value of `"/"` would result in switches that look like **`/ZI`**, **`/nologo`**, **`/W3`**, and so on.
- **`Order`**: A suggestion to a prospective UI client on the relative location of this `Rule` compared to all other rules in the system.
- **`xmlns`**: A standard XML element. You can see three namespaces listed. These attributes correspond to the namespaces for the XML deserialization classes, XML schema, and system namespace, respectively.
- **`DisplayName`**: The name that's shown on the property page UI for the `Rule` node. This value is localized. We created `DisplayName` as a child element of `Rule` rather than as an attribute (like `Name` or `SwitchPrefix`) because of internal localization tool requirements. From an XML perspective, both are equivalent. So, you can just make it an attribute to reduce clutter or leave it as it is.
- **`DataSource`**: This important property tells the project system the location to read and write the property value, and its grouping (explained later). For *`cl.xml`*, these values are:

XML

```
<DataSource Persistence="ProjectFile" ItemType="ClCompile" Label=""
HasConfigurationCondition="true" />
```

- `Persistence="ProjectFile"` tells the project system that all properties for the `Rule` should be written to the project file or the property sheet file (depending on which node was used to spawn the property pages). The other possible value is `"UserFile"`, which will write the value to the *`.user`* file.

- `ItemType="ClCompile"` says that the properties will be stored as ItemDefinition metadata or item metadata (the latter only if the property pages were spawned from a file node in solution explorer) of this item type. If this field isn't set, then the property is written as a common property in a PropertyGroup.

- `Label=""` indicates that when the properties are written as `ItemDefinition` metadata, the label of the parent ItemDefinitionGroup will be empty (every MSBuild element can have a Label). Visual Studio 2017 and later use labeled groups to navigate the .vcxproj project file. The groups that contain most `Rule` properties have an empty string as a label.

- `HasConfigurationCondition="true"` tells the project system to affix a configuration condition to the value so that it takes effect only for the current project configuration (the condition could be affixed to the parent group or the value itself). For example, open the property pages off the project node and set the value of the property **Treat Warnings As Error** under **Configuration Properties** > **C/C++** **General** to "Yes". The following value is written to the project file. Notice the configuration condition attached to the parent ItemDefinitionGroup.

 XML

  ```
  <ItemDefinitionGroup
  Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">
    <ClCompile>
      <TreatWarningAsError>true</TreatWarningAsError>
    </ClCompile>
  </ItemDefinitionGroup>
  ```

 If this value is set in the property page for a specific file, such as *`stdafx.cpp`*, then the property value should be written under the `stdafx.cpp` item in the project file as shown Here. Notice how the configuration condition is directly attached to the metadata itself:

```xml
<ItemGroup>
  <ClCompile Include="stdafx.cpp">
    <TreatWarningAsError
Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">true</Trea
tWarningAsError>
  </ClCompile>
</ItemGroup>
```

Another attribute of `DataSource` not listed here is `PersistedName`. You can use this attribute to represent a property in the project file using a different name. By default, this attribute is set to the property's `Name`.

An individual property can override the `DataSource` of its parent `Rule`. In that case, the location for that property's value will be different from other properties in the `Rule`.

- There are other attributes of a `Rule`, including `Description` and `SupportsFileBatching`, that aren't shown here. The full set of attributes applicable to a `Rule` or on any other element can be obtained by browsing the documentation for these types. Alternately, you can examine the public properties on the types in the `Microsoft.Build.Framework.XamlTypes` namespace in the `Microsoft.Build.Framework.dll` assembly.

- **`DisplayName`**, **`PageTemplate`**, and **`Order`** are UI-related properties that are present in this otherwise UI-independent data model. These properties are almost certain to be used by any UI that is used to display the property pages. `DisplayName` and `Description` are two properties that are present on almost all elements in the XML file. And, these two properties are the only ones that are localized.

Category elements

A `Rule` can have multiple `Category` elements. The order in which the categories are listed in the XML file is a suggestion to the UI to display the categories in the same order. For example, the order of the categories under the **C/C++** node you see in the UI is the same as the order in *`cl.xml`*. A sample category looks like this:

```xml
<Category Name="Optimization">
  <Category.DisplayName>
    <sys:String>Optimization</sys:String>
```

```
  </Category.DisplayName>
</Category>
```

This snippet shows the `Name` and `DisplayName` attributes that have been described before. Once again, there are other attributes a `Category` can have that aren't shown in the example. You can learn about them by reading the documentation or by examining the assemblies using *`ildasm.exe`*.

Property elements

Most of the rule file consists of `Property` elements. They contain the list of all properties in a `Rule`. Each property can be one of the five possible types shown in the basic framework: `BoolProperty`, `EnumProperty`, `IntProperty`, `StringProperty`, and `StringListProperty`. You might have only a few of those types in your file. A property has a number of attributes that allow it to be described in detail. The `StringProperty` is described here. The rest are similar.

XML

```
<StringProperty Subtype="file" Name="ObjectFileName" Category="Output Files"
Switch="Fo">
  <StringProperty.DisplayName>
    <sys:String>Object File Name</sys:String>
  </StringProperty.DisplayName>
  <StringProperty.Description>
    <sys:String>Specifies a name to override the default object file name;
can be file or directory name.(/Fo[name])</sys:String>
  </StringProperty.Description>
</StringProperty>
```

Most of the attributes in the snippet have been described before. The new ones are `Subtype`, `Category`, and `Switch`.

- **`Subtype`** is an attribute available only for `StringProperty` and `StringListProperty` elements. It gives contextual information. For example, the value `file` indicates that the property represents a file path. Visual Studio uses such contextual information to enhance the editing experience. For instance, it may provide a Windows Explorer window that allows the user to choose the file visually as the property's editor.

- **`Category`**: The category under which this property falls. Try to find this property under the **Output Files** category in the UI.

- **`Switch`**: When a rule represents a tool such as the compiler tool, most `Rule` properties get passed as switches to the tool executable at build time. The value of this attribute indicates which switch literal to use. The `<StringProperty>` example specifies that its switch should be **`Fo`**. Combined with the `SwitchPrefix` attribute on the parent `Rule`, this property is passed to the executable as **`/Fo"Debug\"`**. It's visible in the command line for C/C++ in the property page UI.

 Other property attributes include:

- **`Visible`**: If you don't want your property to appear in the property pages, but want it available at build time, set this attribute to `false`.

- **`ReadOnly`**: If you want to provide a read-only view of this property's value in the property pages, set this attribute to `true`.

- **`IncludeInCommandLine`**: At build time, a tool might not need some of its properties. Set this attribute to `false` to prevent a particular property from being passed.

`.vcxproj` and `.props` file structure

Article • 11/15/2022

MSBuild is the default project system in Visual Studio; when you choose **File** > **New Project** in Visual C++ you're creating an MSBuild project whose settings are stored in an XML project file that has the extension `.vcxproj`. The project file may also import `.props` files and `.targets` files where settings can be stored.

If you intend to maintain your project properties in the IDE, we recommend you only create and modify your `.vcxproj` projects in the IDE, and avoid manual edits to the files. In most cases, you never need to manually edit the project file. Manual edits may break the project connections required to modify project settings in the Visual Studio property pages, and can cause build errors that are difficult to debug and repair. For more information about using the property pages, see Set C++ compiler and build properties in Visual Studio.

At scale, managing many individual projects in the IDE becomes tedious and error-prone. It's hard to maintain consistency or enforce standardization across tens or hundreds of projects. In these cases, it's worthwhile to edit your project files to use customized `.props` or `.targets` files for common properties across many projects. You may also use these files when you require customizations that aren't possible in the IDE. Handy places to insert customizations are the `Directory.Build.props` and `Directory.Build.targets` files, which are automatically imported in all MSBuild-based projects.

In some cases, customized `.props` or `.targets` files alone may not be sufficient for your project management needs. You may still need to modify `.vcxproj` project files or property sheets manually. Manual editing requires a good understanding of MSBuild, and must follow the guidelines in this article. In order for the IDE to load and update `.vcxproj` files automatically, these files have several restrictions that don't apply to other MSBuild project files. Mistakes can cause the IDE to crash or behave in unexpected ways.

For manual editing scenarios, this article contains basic information about the structure of `.vcxproj` and related files.

Important considerations

If you choose to manually edit a `.vcxproj` file, be aware of these facts:

- The structure of the file must follow a prescribed form, which is described in this article.

- The Visual Studio C++ project system currently doesn't support wildcards or lists directly in project items. For example, these forms aren't supported:

 XML

  ```
  <ItemGroup>
     <None Include="*.txt"/>
     <ClCompile Include="a.cpp;b.cpp"/>
  </ItemGroup>
  ```

 For more information on wildcard support in projects and possible workarounds, see .vcxproj files and wildcards.

- The Visual Studio C++ project system currently doesn't support macros in project item paths. For example, this form isn't supported:

 XML

  ```
  <ItemGroup>
     <ClCompile Include="$(IntDir)\generated.cpp"/>
  </ItemGroup>
  ```

 "Not supported" means that macros aren't guaranteed to work for all operations in the IDE. Macros that don't change their value in different configurations should work, but might not be preserved if an item is moved to a different filter or project. Macros that change their value for different configurations will cause problems. The IDE doesn't expect project item paths to be different for different project configurations.

- To add, remove, or modify project properties correctly when you edit them in the **Project Properties** dialog, the file must contain separate groups for each project configuration. The conditions must be in this form:

 XML

  ```
  Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'"
  ```

- Each property must be specified in the group with its correct label, as specified in the property rule file. For more information, see Property page xml rule files.

`.vcxproj` file elements

You can inspect the contents of a `.vcxproj` file by using any text or XML editor. You can view it in Visual Studio by right-clicking on the project in Solution Explorer, choosing **Unload project** and then choosing **Edit Foo.vcxproj**.

The first thing to notice is that the top-level elements appear in a particular order. For example:

- Most of the property groups and item definition groups occur after the import for Microsoft.Cpp.Default.props.
- All targets are imported at the end of the file.
- There are multiple property groups, each with a unique label, and they occur in a particular order.

The order of elements in the project file is vitally important, because MSBuild is based on a sequential evaluation model. If your project file, including all the imported `.props` and `.targets` files, consists of multiple definitions of a property, the last definition overrides the preceding ones. In the following example, the value "xyz" will be set during compilation because the MSBUild engine encounters it last during its evaluation.

XML

```
  <MyProperty>abc</MyProperty>
  <MyProperty>xyz</MyProperty>
```

The following snippet shows a minimal `.vcxproj` file. Any `.vcxproj` file generated by Visual Studio will contain these top-level MSBuild elements. And, they'll appear in this order, although they may contain multiple copies of each such top-level element. Any `Label` attributes are arbitrary tags that are only used by Visual Studio as signposts for editing; they have no other function.

XML

```
<Project DefaultTargets="Build" ToolsVersion="4.0"
xmlns='http://schemas.microsoft.com/developer/msbuild/2003'>
  <ItemGroup Label="ProjectConfigurations" />
  <PropertyGroup Label="Globals" />
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.default.props" />
  <PropertyGroup Label="Configuration" />
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />
  <ImportGroup Label="ExtensionSettings" />
  <ImportGroup Label="PropertySheets" />
  <PropertyGroup Label="UserMacros" />
```

```
  <PropertyGroup />
  <ItemDefinitionGroup />
  <ItemGroup />
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />
  <ImportGroup Label="ExtensionTargets" />
</Project>
```

The following sections describe the purpose of each of these elements and why they're ordered this way:

Project element

XML

```
<Project DefaultTargets="Build" ToolsVersion="4.0"
xmlns='http://schemas.microsoft.com/developer/msbuild/2003' >
```

`Project` is the root node. It specifies the MSBuild version to use and also the default target to execute when this file gets passed to MSBuild.exe.

ProjectConfigurations ItemGroup element

XML

```
<ItemGroup Label="ProjectConfigurations" />
```

`ProjectConfigurations` contains the project configuration description. Examples are Debug|Win32, Release|Win32, Debug|ARM and so on. Many project settings are specific to a given configuration. For example, you'll probably want to set optimization properties for a release build but not a debug build.

The `ProjectConfigurations` item group isn't used at build time. The Visual Studio IDE requires it to load the project. This item group can be moved to a *`.props`* file and imported into the *`.vcxproj`* file. However, in that case, if you need to add or remove configurations, you must manually edit the *`.props`* file; you can't use the IDE.

ProjectConfiguration elements

The following snippet shows a project configuration. In this example, 'Debug|x64' is the configuration name. The project configuration name must be in the format `$(Configuration)|$(Platform)`. A `ProjectConfiguration` node can have two properties:

`Configuration` and `Platform`. Those properties get set automatically with the values specified here when the configuration is active.

```
XML

<ProjectConfiguration Include="Debug|x64">
  <Configuration>Debug</Configuration>
  <Platform>x64</Platform>
</ProjectConfiguration>
```

The IDE expects to find a project configuration for any combination of `Configuration` and `Platform` values used in all `ProjectConfiguration` items. Often, it means that a project might have meaningless project configurations to fulfill this requirement. For instance, if a project has these configurations:

- Debug|Win32
- Retail|Win32
- Special 32-bit Optimization|Win32

then it must also have these configurations, even though "Special 32-bit Optimization" is meaningless for x64:

- Debug|x64
- Retail|x64
- Special 32-bit Optimization|x64

You can disable the build and deploy commands for any configuration in the **Solution Configuration Manager**.

Globals PropertyGroup element

```
XML

<PropertyGroup Label="Globals" />
```

`Globals` contains project level settings such as `ProjectGuid`, `RootNamespace`, and `ApplicationType` or `ApplicationTypeRevision`. The last two often define the target OS. A project can only target a single OS because currently, references and project items can't have conditions. These properties are typically not overridden elsewhere in the project

file. This group isn't configuration-dependent, and typically only one `Globals` group exists in the project file.

Microsoft.Cpp.default.props Import element

```XML
<Import Project="$(VCTargetsPath)\Microsoft.Cpp.default.props" />
```

The **Microsoft.Cpp.default.props** property sheet comes with Visual Studio and can't be modified. It contains the default settings for the project. The defaults might vary depending on the ApplicationType.

Configuration PropertyGroup elements

```XML
<PropertyGroup Label="Configuration" />
```

A `Configuration` property group has an attached configuration condition (such as `Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'"`) and comes in multiple copies, one per configuration. This property group hosts the properties that are set for a specific configuration. Configuration properties include PlatformToolset and also control the inclusion of system property sheets in **Microsoft.Cpp.props**. For example, if you define the property `<CharacterSet>Unicode</CharacterSet>`, then the system property sheet **microsoft.Cpp.unicodesupport.props** will be included. If you inspect **Microsoft.Cpp.props**, you'll see the line: `<Import Condition="'$(CharacterSet)' == 'Unicode'" Project="$(VCTargetsPath)\microsoft.Cpp.unicodesupport.props" />`.

Microsoft.Cpp.props Import element

```XML
<Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />
```

The **Microsoft.Cpp.props** property sheet (directly or via imports) defines the default values for many tool-specific properties. Examples include the compiler's Optimization and Warning Level properties, the MIDL tool's TypeLibraryName property, and so on. It also imports various system property sheets based on which configuration properties are defined in the property group immediately before it.

ExtensionSettings ImportGroup element

```XML
<ImportGroup Label="ExtensionSettings" />
```

The `ExtensionSettings` group contains imports for the property sheets that are part of Build Customizations. A Build Customization is defined by up to three files: a *`.targets`* file, a *`.props`* file, and an *`.xml`* file. This import group contains the imports for the *`.props`* file.

PropertySheets ImportGroup elements

```XML
<ImportGroup Label="PropertySheets" />
```

The `PropertySheets` group contains the imports for user property sheets. These imports are the property sheets that you add through the Property Manager view in Visual Studio. The order in which these imports are listed is important and is reflected in the Property Manager. The project file normally contains multiple instances of this kind of import group, one for each project configuration.

UserMacros PropertyGroup element

```XML
<PropertyGroup Label="UserMacros" />
```

`UserMacros` contains properties you create as variables that are used to customize your build process. For example, you can define a user macro to define your custom output path as $(CustomOutputPath) and use it to define other variables. This property group houses such properties. In Visual Studio, this group isn't populated in the project file because Visual C++ doesn't support user macros for configurations. User macros are supported in property sheets.

Per-configuration PropertyGroup elements

```XML
```

```
<PropertyGroup />
```

There are multiple instances of this property group, one per configuration for all project configurations. Each property group must have one configuration condition attached. If any configurations are missing, the **Project Properties** dialog won't work correctly. Unlike the property groups listed before, this one doesn't have a label. This group contains project configuration-level settings. These settings apply to all files that are part of the specified item group. Build customization item definition metadata is initialized here.

This PropertyGroup must come after `<Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />` and there must be no other PropertyGroup without a Label before it (otherwise Project Properties editing won't work correctly).

Per-configuration ItemDefinitionGroup elements

XML

```
<ItemDefinitionGroup />
```

Contains item definitions. These definitions must follow the same conditions rules as the label-less per-configuration `PropertyGroup` elements.

ItemGroup elements

XML

```
<ItemGroup />
```

`ItemGroup` elements contain the items (source files, and so on) in the project. Conditions aren't supported for Project items (that is, item types that are treated as project items by rules definitions).

The metadata should have configuration conditions for each configuration, even if they're all the same. For example:

XML

```
<ItemGroup>
  <ClCompile Include="stdafx.cpp">
```

```
    <TreatWarningAsError
Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">true</TreatWarning
AsError>
    <TreatWarningAsError
Condition="'$(Configuration)|$(Platform)'=='Debug|x64'">true</TreatWarningAs
Error>
  </ClCompile>
</ItemGroup>
```

The Visual Studio C++ project system currently doesn't support wildcards in project items.

XML

```
<ItemGroup>
  <ClCompile Include="*.cpp"> <!--Error-->
</ItemGroup>
```

The Visual Studio C++ project system currently doesn't support macros in project items.

XML

```
<ItemGroup>
  <ClCompile Include="$(IntDir)\generated.cpp"> <!--not guaranteed to work
in all scenarios-->
</ItemGroup>
```

References are specified in an ItemGroup, and they have these limitations:

- References don't support conditions.
- References metadata don't support conditions.

Microsoft.Cpp.targets Import element

XML

```
<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />
```

Defines (directly or through imports) C++ targets such as build, clean, and so on.

ExtensionTargets ImportGroup element

XML

```
<ImportGroup Label="ExtensionTargets" />
```

This group contains imports for the Build Customization target files.

Consequences of incorrect ordering

The Visual Studio IDE depends on the project file having the ordering described previously. For example, when you define a property value in the property pages, the IDE will generally place the property definition in the property group with the empty label. This ordering ensures that default values brought in the system property sheets are overridden by user-defined values. Similarly, the target files are imported at the end since they consume the properties defined before, and since they generally don't define properties themselves. Likewise, user property sheets are imported after the system property sheets (included by `Microsoft.Cpp.props`). This order ensures that the user can override any defaults brought in by the system property sheets.

If a `.vcxproj` file doesn't follow this layout, the build results may not be what you expect. For example, if you mistakenly import a system property sheet after the property sheets defined by the user, the user settings get overridden by the system property sheets.

Even the IDE design time experience depends on some extent on correct ordering of elements. For example, if your `.vcxproj` file doesn't have the `PropertySheets` import group, the IDE might be unable to determine where to place a new property sheet that the user has created in **Property Manager**. It could result in a user sheet being overridden by a system sheet. Although the heuristic used by IDE can tolerate minor inconsistencies in the `.vcxproj` file layout, we strongly recommend you don't deviate from the structure shown earlier in this article.

How the IDE uses element labels

In the IDE, when you set the **UseOfAtl** property in the general property page, it's written to the Configuration property group in the project file. The **TargetName** property in the same property page is written to the label-less per-configuration property group. Visual Studio looks at the property page's xml file for the information on where to write each property. For the **General** property page, assuming you have an English version of Visual Studio 2019 Enterprise Edition, that file is `%ProgramFiles(x86)%\Microsoft Visual Studio\2019\Enterprise\Common7\IDE\VC\VCTargets\1033\general.xml`. The property page XML rule file defines the static information about a Rule and all its properties. One such

piece of information is the preferred position of a Rule property in the destination file (the file where its value will be written). The preferred position is specified by the Label attribute on the project file elements.

Property Sheet layout

The following XML snippet is a minimal layout of a property sheet (.props) file. It's similar to a `.vcxproj` file, and the functionality of the `.props` elements can be inferred from the earlier discussion.

XML

```
<Project ToolsVersion="4.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ImportGroup Label="PropertySheets" />
  <PropertyGroup Label="UserMacros" />
  <PropertyGroup />
  <ItemDefinitionGroup />
  <ItemGroup />
</Project>
```

To make your own property sheet, copy one of the `.props` files in the `VCTargets` folder and modify it for your purposes. For Visual Studio 2019 Enterprise edition, the default `VCTargets` path is `%ProgramFiles%\Microsoft Visual Studio\2019\Enterprise\Common7\IDE\VC\VCTargets`.

See also

Set C++ compiler and build properties in Visual Studio
Property Page XML Files
.vcxproj files and wildcards

`.vcxproj` files and wildcards

Article • 08/03/2021

The Visual Studio IDE doesn't support certain constructs in project items in `.vcxproj` files. These unsupported constructs include wildcards, semi-colon delimited lists, or MSBuild macros that expand to multiple files. The `.vcxproj` project system for C++ builds is more restrictive than MSBuild. Each project item is required to have its own MSBuild item. For more information on the `.vcxproj` file format, see .vcxproj and .props file structure.

These construct examples aren't supported by the IDE:

XML

```
<ItemGroup>
  <None Include="*.txt">
  <ClCompile Include="a.cpp;b.cpp"/>
  <ClCompile Include="@(SomeItems)" />
</ItemGroup>
```

If a `.vcxproj` project file that includes these constructs gets loaded in the IDE, the project may seem to work at first. However, issues are likely as soon as the project is modified by Visual Studio and then saved on disk. You may experience random crashes and undefined behavior.

In Visual Studio 2019 version 16.7, when Visual Studio loads a `.vcxproj` project file, it automatically detects unsupported entries in project items. You'll see warnings in the Output window during solution load.

Visual Studio 2019 version 16.7 also adds read-only project support. Read-only support allows the IDE to use manually authored projects that don't have the additional limitations of IDE-editable projects.

If you have a `.vcxproj` file that uses one or more of the unsupported constructs, you can make it load without warnings in the IDE by using one of these options:

- List all items explicitly
- Mark your project as read-only
- Move wildcard items to a target body

List all items explicitly

Currently, there's no way to make wildcard expansion items visible in the Solution Explorer window in a non-read-only project. Solution Explorer expects projects to list all items explicitly.

To make `.vcxproj` projects automatically expand wildcards in Visual Studio 2019 version 16.7 or later, set the `ReplaceWildcardsInProjectItems` property to `true`. We recommend you create a *`Directory.Build.props`* file in a root directory, and use this content:

XML

```xml
<Project>
  <PropertyGroup>
    <ReplaceWildcardsInProjectItems>true</ReplaceWildcardsInProjectItems>
  </PropertyGroup>
</Project>
```

Mark your project as read-only

In Visual Studio 2019 version 16.7 and later, you can mark projects as *read-only*. To mark your project read-only, add the following property to your *`.vcxproj`* file, or to any of the files it imports:

XML

```xml
<PropertyGroup>
    <ReadOnlyProject>true</ReadOnlyProject>
</PropertyGroup>
```

The `<ReadOnlyProject>` setting prevents Visual Studio from editing and saving the project, so you can use any MSBuild constructs in it, including wildcards.

It's important to know that the project cache isn't available if Visual Studio detects wildcards in project items in the *`.vcxproj`* file or any of its imports. Solution load times in the IDE are much longer if you have lots of projects that use wildcards.

Move wildcard items to a target body

You may want to use wildcards to gather resources, add generated sources, and so on. If you don't need them listed in the Solution Explorer window, you can use this procedure instead:

1. Change the name of the item group to add wildcards. For instance, instead of:

```xml
<Image Include="*.bmp" />
<ClCompile Include="*.cpp" />
```

change it to:

```xml
<_WildCardImage Include="*.bmp" />
<_WildCardClCompile Include="*.cpp" />
```

2. Add this content to your `.vcxproj` file. Or, add it to a `Directory.Build.targets` file in a root directory, to affect all projects under that root:

```xml
<Target Name="AddWildCardItems"
    AfterTargets="BuildGenerateSources">
  <ItemGroup>
    <Image Include="@(_WildCardImage)" />
    <ClCompile Include="@(_WildCardClCompile)" />
  </ItemGroup>
</Target>
```

 This change makes the build see the items as they're defined in the `.vcxproj` file. However, now they aren't visible in the Solution Explorer window, and they won't cause problems in the IDE.

3. To show correct IntelliSense for `_WildCardClCompile` items when you open those files in the editor, add the following content:

```xml
<PropertyGroup>
  <ComputeCompileInputsTargets>
    AddWildCardItems
    $(ComputeCompileInputsTargets)
  </ComputeCompileInputsTargets>
</PropertyGroup>
```

Effectively, you can use wildcards for any items inside a target body. You can also use wildcards in an `ItemGroup` that isn't defined as a project item by a ProjectSchemaDefinition.

> **Note**
>
> If you move wildcard includes from a `.vcxproj` file to an imported file, they won't be visible in the Solution Explorer window. This change also allows your project to load in the IDE without modification. However, we don't recommend this approach, because it disables the project cache.

See also

Set C++ compiler and build properties in Visual Studio
Property Page XML Files

Project Files

Article • 08/03/2021

A C++ project file in Visual Studio is an XML-based file that has the .vcxproj file name extension and contains information that is required to build a C++ project. Note that the project file imports various project files that have the ".props" or ".targets" extension. These files contain additional build information, and might themselves refer to other ".props" or ".targets" files. The macros in the file path (for example `$(VCTargetsPath)`) are dependent on your Visual Studio installation. For more information about these macros and ".props" and ".targets" files, see VC++ Directories Property Page, Set C++ compiler and build properties in Visual Studio and Common macros for build commands and properties.

Example

The following sample .vcxproj file was produced by choosing **Windows Desktop Wizard** in the **New Project** dialog box. To process a project file use either the msbuild.exe tool at the command line, or the **Build** command in the IDE. (This sample cannot be processed because the required source and header files are not provided.) For more information about the XML elements in a project file, see Project File Schema Reference.

> **Note**
>
> For projects in Visual Studio 2017 and earlier, change `pch.h` to `stdafx.h` and `pch.cpp` to `stdafx.cpp`.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" ToolsVersion="4.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup Label="ProjectConfigurations">
    <ProjectConfiguration Include="Debug|Win32">
      <Configuration>Debug</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
    <ProjectConfiguration Include="Release|Win32">
      <Configuration>Release</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
  </ItemGroup>
  <PropertyGroup Label="Globals">
    <ProjectGuid>{96F21549-A7BF-4695-A1B1-B43625B91A14}</ProjectGuid>
```

```
    <Keyword>Win32Proj</Keyword>
    <RootNamespace>SomeProjName</RootNamespace>
  </PropertyGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'"
Label="Configuration">
    <ConfigurationType>Application</ConfigurationType>
    <CharacterSet>Unicode</CharacterSet>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|Win32'"
Label="Configuration">
    <ConfigurationType>Application</ConfigurationType>
    <WholeProgramOptimization>true</WholeProgramOptimization>
    <CharacterSet>Unicode</CharacterSet>
  </PropertyGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />
  <ImportGroup Label="ExtensionSettings">
  </ImportGroup>
  <ImportGroup Label="PropertySheets"
Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">
    <Import Project="$(UserRootDir)\Microsoft.Cpp.$(Platform).user.props"
Condition="exists('$(UserRootDir)\Microsoft.Cpp.$(Platform).user.props')"
Label="LocalAppDataPlatform" />
  </ImportGroup>
  <ImportGroup Label="PropertySheets"
Condition="'$(Configuration)|$(Platform)'=='Release|Win32'">
    <Import Project="$(UserRootDir)\Microsoft.Cpp.$(Platform).user.props"
Condition="exists('$(UserRootDir)\Microsoft.Cpp.$(Platform).user.props')"
Label="LocalAppDataPlatform" />
  </ImportGroup>
  <PropertyGroup Label="UserMacros" />
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">
    <LinkIncremental>true</LinkIncremental>
  </PropertyGroup>
  <PropertyGroup
Condition="'$(Configuration)|$(Platform)'=='Release|Win32'">
    <LinkIncremental>false</LinkIncremental>
  </PropertyGroup>
  <ItemDefinitionGroup
Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">
    <ClCompile>
      <PrecompiledHeader>Use</PrecompiledHeader>
      <WarningLevel>Level3</WarningLevel>
      <MinimalRebuild>true</MinimalRebuild>
      <DebugInformationFormat>EditAndContinue</DebugInformationFormat>
      <Optimization>Disabled</Optimization>
      <BasicRuntimeChecks>EnableFastChecks</BasicRuntimeChecks>
      <RuntimeLibrary>MultiThreadedDebugDLL</RuntimeLibrary>
      <PreprocessorDefinitions>WIN32;_DEBUG;_CONSOLE;%
(PreprocessorDefinitions)</PreprocessorDefinitions>
    </ClCompile>
    <Link>
      <SubSystem>Console</SubSystem>
      <GenerateDebugInformation>true</GenerateDebugInformation>
    </Link>
```

```
  </ItemDefinitionGroup>
  <ItemDefinitionGroup
Condition="'$(Configuration)|$(Platform)'=='Release|Win32'">
    <ClCompile>
      <WarningLevel>Level3</WarningLevel>
      <PrecompiledHeader>Use</PrecompiledHeader>
      <DebugInformationFormat>ProgramDatabase</DebugInformationFormat>
      <Optimization>MaxSpeed</Optimization>
      <RuntimeLibrary>MultiThreadedDLL</RuntimeLibrary>
      <FunctionLevelLinking>true</FunctionLevelLinking>
      <IntrinsicFunctions>true</IntrinsicFunctions>
      <PreprocessorDefinitions>WIN32;NDEBUG;_CONSOLE;%
(PreprocessorDefinitions)</PreprocessorDefinitions>
    </ClCompile>
    <Link>
      <SubSystem>Console</SubSystem>
      <GenerateDebugInformation>true</GenerateDebugInformation>
      <EnableCOMDATFolding>true</EnableCOMDATFolding>
      <OptimizeReferences>true</OptimizeReferences>
    </Link>
  </ItemDefinitionGroup>
  <ItemGroup>
    <None Include="ReadMe.txt" />
  </ItemGroup>
  <ItemGroup>
    <ClInclude Include="pch.h" />
    <ClInclude Include="targetver.h" />
  </ItemGroup>
  <ItemGroup>
    <ClCompile Include="SomeProjName.cpp" />
    <ClCompile Include="pch.cpp">
      <PrecompiledHeader
Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'">Create</Precompile
dHeader>
      <PrecompiledHeader
Condition="'$(Configuration)|$(Platform)'=='Release|Win32'">Create</Precompi
ledHeader>
    </ClCompile>
  </ItemGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />
  <ImportGroup Label="ExtensionTargets">
  </ImportGroup>
</Project>
```

See also

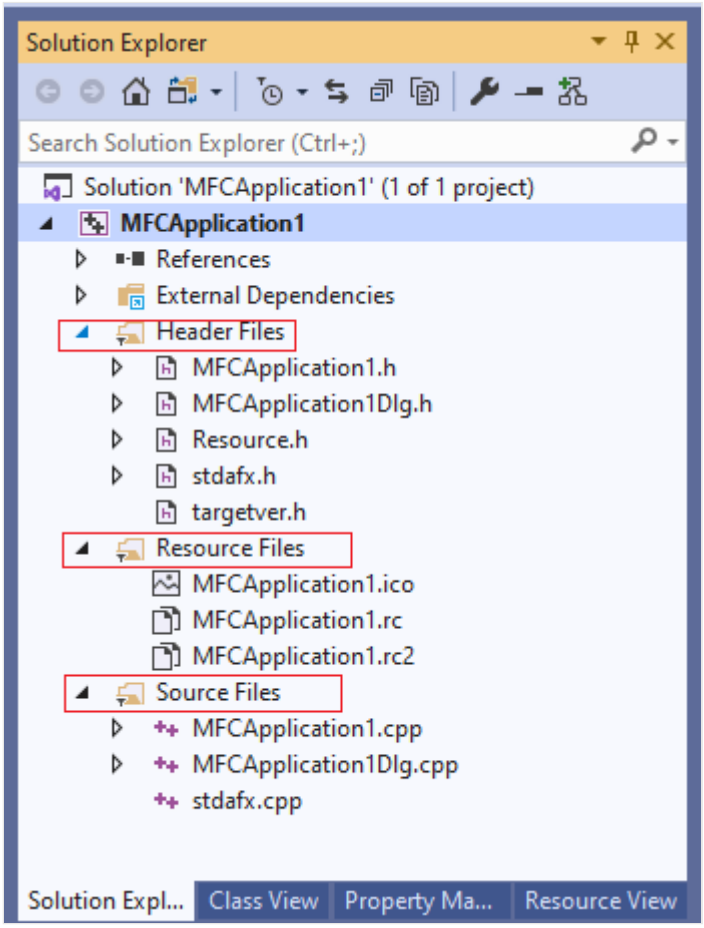
Visual Studio Projects - C++

Set C++ compiler and build properties in Visual Studio

vcxproj.filters files

Article • 04/10/2023

The *filters* file (`*.vcxproj.filters`) is an XML file in MSBuild format that is located in the root project folder. It specifies which file types go into which logical folder in **Solution Explorer**. In the following illustration, the `.cpp` files are under the **Source Files** node. the `.h` files are under the **Header Files** node, and `.ico` and `.rc` files are under **Resource Files**. This placement is controlled by the filters file.



Creating a custom filters file

Visual Studio creates this file automatically. For desktop applications, the predefined logical folders (filters) are: **Source Files**, **Header Files** and **Resource Files**. Other project types such as UWP might have a different set of default folders. Visual Studio automatically assigns known file types to each folder. If you want to create a filter with a custom name or a filter that holds custom file types, you can create your own filters file in the root folder of the project, or under an existing filter. (**References** and **External Dependencies** are special folders that don't participate in filtering.)

Example

The following example shows the filters file for the example show previously. It has a flat hierarchy; in other words, there are no nested logical folders. The `UniqueIdentifier` node is optional. It enables Visual Studio automation interfaces to find the filter. `Extensions` is also optional. When a new file is added to a project, it's added to the topmost filter with a matching file extension. To add a file to a specific filter, right-click on the filter and choose **Add New Item**.

The `ItemGroup` that contains the `ClInclude` nodes is created when the project is first launched. If you're generating your own vcxproj files, make sure that all project items also have an entry in the filters file. Values in a `ClInclude` node override the default filtering based on file extensions. When you use Visual Studio to add a new item to the project, the IDE adds an individual file entry in the filters file. The filter isn't automatically reassigned if you change the file's extension.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <Filter Include="Source Files">
      <UniqueIdentifier>{4FC737F1-C7A5-4376-A066-2A32D752A2FF}
</UniqueIdentifier>
      <Extensions>cpp;c;cc;cxx;def;odl;idl;hpj;bat;asm;asmx</Extensions>
    </Filter>
    <Filter Include="Header Files">
      <UniqueIdentifier>{93995380-89BD-4b04-88EB-625FBE52EBFB}
</UniqueIdentifier>
      <Extensions>h;hh;hpp;hxx;hm;inl;inc;ipp;xsd</Extensions>
    </Filter>
    <Filter Include="Resource Files">
      <UniqueIdentifier>{67DA6AB6-F800-4c08-8B7A-83BB121AAD01}
</UniqueIdentifier>

<Extensions>rc;ico;cur;bmp;dlg;rc2;rct;bin;rgs;gif;jpg;jpeg;jpe;resx;tiff;ti
f;png;wav;mfcribbon-ms</Extensions>
    </Filter>
  </ItemGroup>
  <ItemGroup>
    <ClInclude Include="MFCApplication1.h">
      <Filter>Header Files</Filter>
    </ClInclude>
    <ClInclude Include="MFCApplication1Dlg.h">
      <Filter>Header Files</Filter>
    </ClInclude>
    <ClInclude Include="stdafx.h">
      <Filter>Header Files</Filter>
```

```
    </ClInclude>
    <ClInclude Include="targetver.h">
      <Filter>Header Files</Filter>
    </ClInclude>
    <ClInclude Include="Resource.h">
      <Filter>Header Files</Filter>
    </ClInclude>
  </ItemGroup>
  <ItemGroup>
    <ClCompile Include="MFCApplication1.cpp">
      <Filter>Source Files</Filter>
    </ClCompile>
    <ClCompile Include="MFCApplication1Dlg.cpp">
      <Filter>Source Files</Filter>
    </ClCompile>
    <ClCompile Include="stdafx.cpp">
      <Filter>Source Files</Filter>
    </ClCompile>
  </ItemGroup>
  <ItemGroup>
    <ResourceCompile Include="MFCApplication1.rc">
      <Filter>Resource Files</Filter>
    </ResourceCompile>
  </ItemGroup>
  <ItemGroup>
    <None Include="res\MFCApplication1.rc2">
      <Filter>Resource Files</Filter>
    </None>
  </ItemGroup>
  <ItemGroup>
    <Image Include="res\MFCApplication1.ico">
      <Filter>Resource Files</Filter>
    </Image>
  </ItemGroup>
</Project>
```

To create nested logical folders, declare all nodes in filters `ItemGroup` as shown below. Each child node must declare the full logical path to the topmost parent. In the following example, an empty `ParentFilter` must be declared because it's referenced in later nodes.

XML

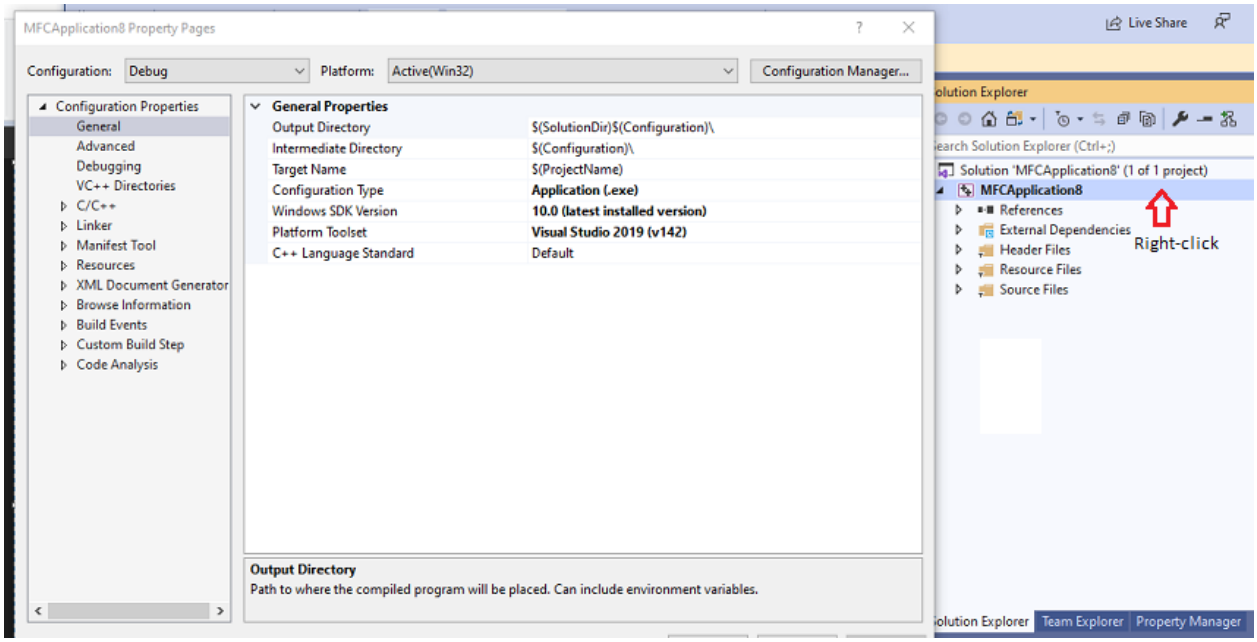
```
  <ItemGroup>
    <Filter Include="ParentFilter">
    </Filter>
    <Filter Include="ParentFilter\Source Files"> <!-- Full path to topmost
parent.-->
      <UniqueIdentifier>{4FC737F1-C7A5-4376-A066-2A32D752A2FF}
</UniqueIdentifier> <!--  Optional-->
      <Extensions>cpp;c;cc;cxx;def;odl;idl;hpj;bat;asm;asmx</Extensions> <!-
- Optional -->
```

```
    </Filter>
    <Filter Include="Header Files">
      <UniqueIdentifier>{93995380-89BD-4b04-88EB-625FBE52EBFB}
</UniqueIdentifier>
      <Extensions>h;hh;hpp;hxx;hm;inl;inc;ipp;xsd</Extensions>
    </Filter>
  </ItemGroup>
```

Windows C++ project property page reference

Article • 08/03/2021

In Visual Studio, you specify compiler and linker options, file paths, and other build settings through the property pages for the project. The properties and property pages that are available depend on the project type. For example, a makefile project has an NMake property page, which is not present in an MFC or Win32 console project. To open the **Property Pages**, choose **Project** > **Properties** from the main menu, or right-click on the project node in **Solution Explorer** and choose **Properties**. Individual files also have property pages that enable you to set compile and build options for just that file. The following image shows the property pages for an MFC project.



This section provides a quick reference for the property pages themselves. The options and settings exposed in the property pages are documented more completely in their own topics and are linked from the property page topics. For more information about project properties, see Set C++ compiler and build properties in Visual Studio.

For property pages in Linux projects, see Linux C++ Property Page Reference.

In This Section

- General Property Page (Project)
- General Property Page (File)
- Advanced Property Page
- VC++ Directories Property Page

- Linker Property Pages
- Manifest Tool Property Pages
- HLSL Property Pages
- Command Line Property Pages
- Custom Build Step Property Page: General
- Adding references
- Managed Resources Property Page
- MIDL Property Pages
- NMake Property Page
- Resources Property Pages
- Web References Property Page
- XML Data Generator Tool Property Page
- XML Document Generator Tool Property Pages

See also

How to: Create and Remove Project Dependencies
How to: Create and Edit Configurations
Linux C++ Property Page Reference

General Property Page (Project)

Article • 10/06/2021

This article applies to Visual Studio projects for Windows. For Linux projects, see Linux C++ Property page reference. For CMake projects, see CMake projects in Visual Studio. For Android projects, see General project properties (Android C++). For Android Makefile projects, see General project properties (Android C++ Makefile). In Visual Studio 2019, some properties for non-UWP (Windows Runtime or Universal Windows Platform) projects have moved to the Advanced property page.

To open the Property Pages dialog for a project, select the project (not the solution) in Solution Explorer. Next, select the **Project** > **Project-name Properties** menu from the menu bar. Or, right-click on the project node in Solution Explorer and select **Properties** from the shortcut menu.

In the Property Pages dialog, the **Configuration Properties** > **General** property page displays project properties based on project type. These properties are gathered under one or two headings, depending on project type:

- General
- Project Defaults

General

The General property heading includes some combination of these properties:

Target Platform

Specifies the platform that the project runs on. For example, Windows, Android, or iOS. The value **Windows 10** means the project targets the Universal Windows Platform. If you're targeting other versions of Windows, the version isn't listed and the value in this field appears as just **Windows**. This property is a read-only field that's set when you create a project.

Target Platform Version

Specifies the version of the Windows SDK used to build the project. This property appears only if the project type supports it. You can select 10.0 to specify the latest version of the Windows SDK. If your app can take advantage of features in this Windows SDK version, but can still run on earlier versions without those features, perhaps with

some loss of functionality, then the value of this property and the **Target Platform Min. Version** property might be different. If so, your code should check the version of the platform it's running against at runtime and disable features that aren't available in older platform versions.

Target Platform Min. Version

Specifies the lowest version of the platform that the project can run on. This property appears only if the project type supports it. Set this value if your app can take advantage of features in a newer Windows SDK version, but still runs on earlier versions, perhaps with some loss of functionality. If set to a lower value, your code should check the version of the platform it's running against at runtime. Then, disable features that aren't available in older platform versions.

The C++ project system doesn't enforce this option. It's included for consistency with other languages, such as C# and JavaScript, and as a guide for anyone who uses your project. Microsoft C++ won't generate an error if you use a feature that's not available in the minimum version.

Windows SDK Version

For the Windows target platform, this property specifies the version of the Windows SDK that your project requires. When the Visual Studio Installer installs a C++ Workload, it also installs the required parts of the Windows SDK. If you have other Windows SDK versions on your computer, each version installed appears in the dropdown.

To target Windows 7 or Windows Vista, use the value **8.1**, since Windows SDK 8.1 is backward compatible to those platforms. When you target an earlier version, define the appropriate value for `_WIN32_WINNT` in *`targetver.h`*. For Windows 7, that's 0x0601. For more information, see Modifying WINVER and _WIN32_WINNT.

You can install the Windows XP platform toolset included as an optional component in Visual Studio Installer to build Windows XP and Windows 2003 Server projects. For information on how to obtain and use this platform toolset, see Configuring programs for Windows XP. For more information on changing the platform toolset, see How to: Modify the target framework and platform toolset.

Output Directory

Specifies the directory where build tools such as the linker place all final output files created during the build process. Typically, this directory holds the output of tools such

as the linker, librarian, or BSCMake. By default, this property is the directory specified by the macro combination `$(SolutionDir)$(Configuration)\`.

To programmatically access this property, see OutputDirectory.

Intermediate Directory

Specifies the directory where tools such as the compiler place all intermediate files created during the build process. Typically, this directory holds the output of tools such as the C/C++ compiler, MIDL, and the resource compiler. By default, this property is the directory specified by the macro `$(Configuration)\`.

To programmatically access this property, see IntermediateDirectory.

Target Name

Specifies the file name this project generates. By default, this property is the filename specified by the macro `$(ProjectName)`.

Target Extension

Specifies the file extension this project generates, such as *`.exe`* or *`.dll`*. For some Visual Studio 2019 project types, this property has moved to the Advanced property page.

Extensions to Delete on Clean

The **Build** > **Clean** menu command deletes files from the intermediate directory where a project's configuration is built. The build system deletes files that have the specified extensions when you run the **Clean** command or when you rebuild. The build system also deletes any known output of the build no matter where it's located. Deleted files include any intermediate outputs such as *`.obj`* files. Use semicolons (`;`) to separate extensions. You can specify wildcard characters (`*`, `?`) in the extensions.

To programmatically access this property, see DeleteExtensionsOnClean. For some Visual Studio 2019 project types, this property has moved to the Advanced property page.

Build Log File

Allows you to specify a non-default location for the log file that's created whenever you build a project. The default location is specified by the macro combination

`$(IntDir)$(MSBuildProjectName).log`. For some Visual Studio 2019 project types, this property has moved to the Advanced property page.

You can use project macros to change the directory location. For more information, see Common macros for build commands and properties.

Platform Toolset

Specifies the toolset used for building the current configuration. This property allows the project to target a different version of the Visual C++ libraries and compiler. By default, Visual Studio C++ projects target the latest toolset installed by Visual Studio. You can choose one of the toolsets installed by several previous versions of Visual Studio instead. Some older toolsets can create executables that run on Windows XP or Vista. For more information on how to change the platform toolset, see How to: Modify the target framework and platform toolset.

Enable Managed Incremental Build

For managed projects, this property enables detection of external visibility when you generate assemblies. If a change to a managed project isn't visible to other projects, then dependent projects don't get rebuilt. This option can dramatically improve build times in solutions that include managed projects. In Visual Studio 2019 projects, this property has moved to the Advanced property page.

Configuration Type

Specifies the project output and its required tools. In UWP projects, this property appears under the **Project Defaults** heading. There are several configuration types from which to choose, depending on your project type:

Application (.exe)

Displays the linker toolset: The C/C++ Compiler, MIDL, Resource Compiler, Linker, BSCMake, XML Web Service Proxy Generator, custom build, prebuild, prelink, and postbuild events.

Dynamic Library (.dll)

Displays the linker toolset, specifies the `/DLL` linker option, and adds the `_WINDLL` preprocessor definition to the CL command line.

Makefile

Displays the makefile toolset (NMake).

Static Library (.lib)

Displays the librarian toolset. It's the same as the linker toolset, except it replaces the linker with the librarian and omits XML Web Service Proxy Generator.

Utility

Displays the utility toolset (MIDL, custom build, prebuild, and postbuild events).

To programmatically access this property, see ConfigurationType.

C++ Language Standard

Specifies which C++ language standard to use. The default is `/std:c++14`. Specify `/std:c++17` to use C++17 features, `/std:c++20` to use C++20 features, and `/std:c++latest` to use proposed C++23 features or other experimental features. For more information, see /std (Specify language standard version).

C Language Standard

Specifies which C language standard to use. The default is Legacy MSVC, which implements C89, some of C99, and Microsoft-specific extensions. Specify `/std:c11` to use C11 features, and `/std:c17` to use C17 features. For more information, see /std (Specify language standard version)

Project Defaults

Use of MFC

Specifies whether the MFC project statically or dynamically links to the MFC DLL. Non-MFC projects select **Use Standard Windows Libraries**. In Visual Studio 2019 projects, this property has moved to the Advanced property page.

To programmatically access this property, see useOfMfc.

Character Set

Specifies whether the `_UNICODE` or `_MBCS` preprocessor macro should be set. Also affects the linker entry point, where appropriate. In Visual Studio 2019 projects, this property has moved to the Advanced property page.

To programmatically access this property, see CharacterSet.

Common Language Runtime support

Causes the /clr compiler option to be used. In Visual Studio 2019 projects, this property has moved to the Advanced property page.

To programmatically access this property, see ManagedExtensions.

.NET Target Framework Version

In managed projects, specifies the .NET framework version to target. In Visual Studio 2019 projects, this property has moved to the Advanced property page.

Whole Program Optimization

Specifies the /GL compiler option and /LTCG linker option. By default, this property is disabled for Debug configurations, and enabled for Release configurations. In Visual Studio 2019 projects, this property has moved to the Advanced property page.

Windows Store App Support

Specifies whether this project supports Windows Runtime (Universal Windows Platform or UWP) apps. For more information, see /ZW (Windows Runtime Compilation), and the Windows Developer UWP documentation.

Windows Desktop Compatible

Enables the output of this Windows Runtime project to also support desktop apps. This property sets the `<DesktopCompatible>` value in the project file. The **Windows Desktop Compatible** property is available starting in Visual Studio 2019 version 16.9.

See also

C++ project property page reference

General Property Page (File)

Article • 03/03/2022

This topic applies to Windows projects. For non-Windows projects, see Linux C++ Property Page Reference.

When you right-click on a file node **Solution Explorer**, the **General** property page under the **Configuration Properties** node opens. It contains the following properties:

- **Excluded From Build**

 Specifies whether the file should be in the build for the current configuration.

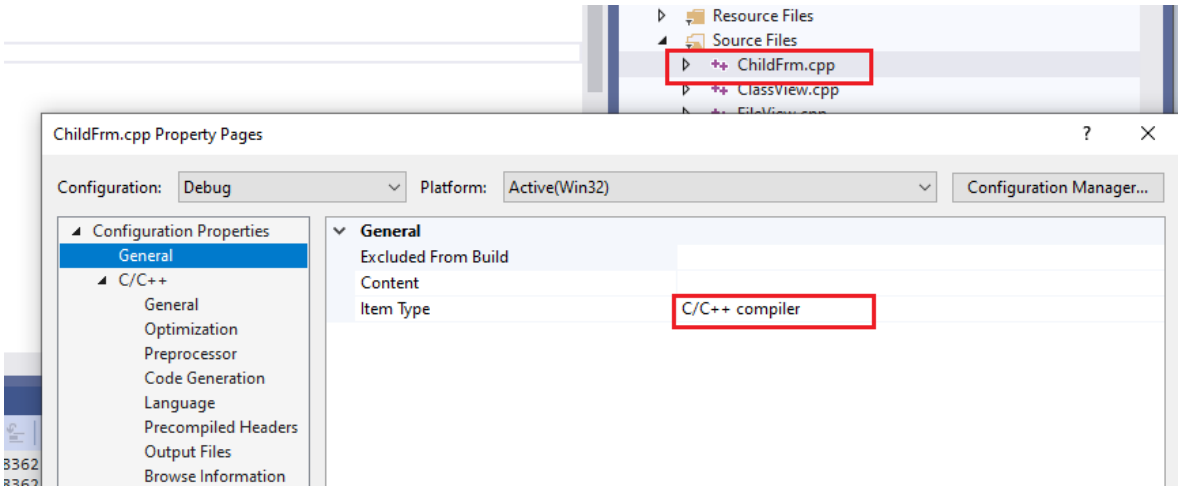
 To programmatically access this property, see ExcludedFromBuild.

- **Content** (Applies to UWP apps only.) Specifies whether the file contains content to be included in the app package.

- **Item Type**

 The **Item Type** specifies the tool that will be used to process the file during the build process. Files whose extension is known to Visual Studio have a default value in this property. You can specify a custom tool here if you have a custom file type or wish to override the default tool for a known file type. For more information, see Specifying Custom Build Tools. You can also use this property page to specify that a file is not part of the build process.

 The following illustration shows the property page for a *.cpp* file. The default **Item Type** for this kind of file is the **C/C++ Compiler** (*cl.exe*) and the property page exposes various compiler settings that can be applied to this file only.



 The following table lists the default Item Types:

File extension	Item Type	Default Tool
.appx	XAML Application Definition	App packager
.hlsl, .cso	HLSL Compiler	fxc.exe
.h	C/C++ Header	C/C++ Preprocessor
n/a	Does not participate in build	n/a
.xml, .xslt, .xsl	Xml	XML Editor
.resw, .resjson	PRI Resource (UWP Apps)	MakePri.exe
	Media (UWP)	App packager
.xsd	XML Data Generator Tool	XML Schema Definition Tool (Xsd.exe) (Requires .NET workload. Not included with MSVC.)
	Manifest Tool	mt.exe
.rc	Resource	Windows Resource Compiler (rc.exe)
.appxmanifest	App Package Manifest	App packager
.obj	Object	C/C++ Linker (link.exe)
.ttf	Font	n/a
.txt	Text	n/a
n/a	Custom Build Tool	User-defined
n/a	Copy file	n/a
.packagelayout	App Package Layout	App packager
.resx	Compiler Managed Resource	Resgen.exe (Resource File Generator)
.natvis	C++ Debugger visualization file	Natvis framework
.jpg, .bmp, .ico, etc.	Image	Resource compiler based on application type.

File extension	Item Type	Default Tool
.cpp	C/C++ Compiler	cl.exe

To programmatically access this property, see Tool.

For information on how to access the **General** property page under the **Configuration Properties** node, see Set C++ compiler and build properties in Visual Studio.

See also

C++ project property page reference

Advanced Property Page

Article • 07/26/2023

The Advanced property page is available in Visual Studio 2019 and later. The specific properties shown depend on the project type. Windows Runtime (Universal Windows Platform, or UWP) projects don't show this page.

Advanced Properties

Target File Extension

Specifies the file extension to use for the build output. Defaults to `.exe` for applications, `.lib` for static libraries and `.dll` for DLLs.

Extensions to Delete on Clean

The **Build** > **Clean** menu command deletes files from the intermediate directory where a project's configuration is built. The build system deletes files that have the specified extensions when you run the **Clean** command or when you rebuild. The build system also deletes any known output of the build no matter where it's located. Deleted files include any intermediate outputs such as `.obj` files. Use semicolons (`;`) to separate extensions. You can specify wildcard characters (`*`, `?`) in the extensions.

To programmatically access this property, see DeleteExtensionsOnClean.

Build Log File

Allows you to specify a non-default location for the log file that's created whenever you build a project. The default location is specified by the macros `$(IntDir)$(MSBuildProjectName).log`.

You can use project macros to change the directory location. For more information, see Common macros for build commands and properties.

Preferred Build Tool Architecture

Specifies whether to use the x86 or x64 build tools.

Use Debug Libraries

Specifies whether to create a Debug or Release build. Despite the name, **Use Debug Libraries** is a build system-specific property that is effectively shorthand for "Make a Debug build" or "Make a Release build". It sets several compiler and linker properties for Debug or Release builds, including the library settings. You can use it to create Debug or Release configurations for a new platform or in a new template. We don't recommend you change this property in an existing configuration. Use the individual compiler and linker properties instead.

Enable Unity (JUMBO) build

Enables a faster build process that combines many C++ source files into one or more files before compilation. These combined files are known as *unity* files. They're unrelated to the Unity game engine.

Copy Content to OutDir

Copy the items marked as *content* in the project to the project's output directory (`$(OutDir)`). This setting can simplify deployment. This property is available starting in Visual Studio 2019 version 16.7.

Copy Project References to OutDir

Copy the executable (DLL and EXE file) project reference items to the project's output directory (`$(OutDir)`). In C++/CLI (/clr) projects, this property is ignored. Instead, the **Copy Local** property on each project reference controls whether it's copied to the output directory. This setting can simplify local deployment. It's available starting in Visual Studio 2019 version 16.7.

Copy Project References' Symbols to OutDir

Copy the PDB files for project reference items along with the project reference executable items to the project's output directory (`$(OutDir)`). This property is always enabled for C++/CLI projects. This setting can simplify debug deployment. It's available starting in Visual Studio 2019 version 16.7.

Copy C++ Runtime to OutDir

Copy the runtime DLLs to the project's output directory (`$(OutDir)`). This setting can simplify local deployment. It's available starting in Visual Studio 2019 version 16.7.

Use of MFC

Specifies whether the MFC project statically or dynamically links to the MFC DLL. Non-MFC projects select **Use Standard Windows Libraries**.

To programmatically access this property, see useOfMfc.

Character Set

Specifies whether the `_UNICODE` or `_MBCS` preprocessor macro should be set. Also affects the linker entry point, where appropriate.

To programmatically access this property, see CharacterSet.

Whole Program Optimization

Specifies the /GL compiler option and /LTCG linker option. By default, this property is disabled for Debug configurations, and enabled for Release configurations.

MSVC Toolset Version

Specifies the full version of the MSVC toolset that's used to build the project. You may have various update and preview versions of a toolset installed. You can specify which one to use here.

LLVM Toolset Version

Specifies the full version of the LLVM toolset that's used to build the project. This property is available when **LLVM (clang-cl)** is selected as the platform toolset, starting in Visual Studio 2019 version 16.9. For more information, see Set a custom LLVM toolset version.

C++/CLI Properties

Common Language Runtime support

Causes the /clr compiler option to be used.

To programmatically access this property, see ManagedExtensions.

.NET Target Framework Version

This property only applies when the **Common Language Runtime support** property is set to **.NET Framework Runtime Support**, that is the project targets .NET Framework, and it specifies the version of the .NET Framework.

.NET Target Framework

This property only applies when the **Common Language Runtime support** property is set to **.NET Runtime Support**, that is the project targets .NET.

This property specifies the .NET 5+ Target Framework Moniker this project targets, for example `net6.0-windows` or `net7.0-windows8.0`.

Enable Managed Incremental Build

For managed projects, this option enables detection of external visibility when you generate assemblies. If a change to a managed project isn't visible to other projects, dependent projects aren't rebuilt. Managed incremental builds can dramatically improve build times in solutions that include managed projects.

Enable CLR Support for Individual Files

This option sets a `ManagedAssembly` build property that enables building only some files in the project as managed code. You must set **Enable CLR Support for Individual Files** to **Yes** if some but not all of your project files are built as managed code. This property is only available in projects that use the v143 or later toolset in Visual Studio 2022 and later versions.

.NET Target Windows Version

This property only applies when the **Common Language Runtime support** property is set to **.NET Runtime Support**, that is the project targets .NET.

This property specifies the minimum Windows version that the project supports. This value is used by NuGet to determine the compatibility of projects and NuGet package dependencies. If a project A depends on project B, project A's .NET target Windows version must be greater or equal to project B's.

C++ Debugging Property Pages

Article • 08/03/2021

These property pages are found under **Project** > **Properties** > **Configuration Properties** > **Debugging**. Choose the debugger type in the drop-down control. For more information about debugging C++ code, see Tutorial: Learn to debug C++ code using Visual Studio and Debugging Native Code.

Local Windows Debugger Property Page

Command

The debug command to execute.

Command Arguments

The command line arguments to pass to the application.

Working Directory

The application's working directory. By default, the directory containing the project file.

Attach

Specifies whether the debugger should attempt to attach to an existing process when debugging starts.

Debugger Type

Specifies the debugger type to use. When set to Auto, the debugger type will be selected based on contents of the exe file.

Choices

- **Native Only** - Native Only
- **Managed Only** - Managed Only
- **Mixed** - Mixed
- **Auto** - Auto
- **Script** - Script

- **GPU Only (C++ AMP)** - GPU Only (C++ AMP)

Environment

Specifies the environment for the program to be debugged, or variables to merge with existing environment.

Debugging Accelerator Type

The debugging accelerator type to use for debugging the GPU code. (Available when the GPU debugger is active.)

GPU Default Breakpoint Behavior

Sets how often the GPU debugger breaks.

Choices

- **Break once per warp** - Break once per warp
- **Break for every thread (like CPU behavior)** - Break for every thread (like CPU behavior)

Merge Environment

Merge specified environment variables with existing environment.

SQL Debugging

Attach the SQL debugger.

Amp Default Accelerator

Override C++ AMP's default accelerator selection. Property does not apply when debugging managed code.

Remote Windows Debugger Property Page

For more information about remote debugging, see Remote Debugging a Visual C++ Project in Visual Studio.

Remote Command

The debug command to execute.

Remote Command Arguments

The command line arguments to pass to the application.

Working Directory

The application's working directory. By default, the directory containing the project file.

Remote Server Name

Specifies a remote server name.

Connection

Specifies the connection type.

Choices

- **Remote with Windows authentication** - Remote with Windows authentication.
- **Remote with no authentication** - Remote with no authentication.

Debugger Type

Specifies the debugger type to use. When set to Auto, the debugger type will be selected based on contents of the exe file.

Choices

- **Native Only** - Native Only
- **Managed Only** - Managed Only
- **Mixed** - Mixed
- **Auto** - Auto
- **Script** - Script
- **GPU Only (C++ AMP)** - GPU Only (C++ AMP)

Environment

Specifies the environment for the program to be debugged, or variables to merge with existing environment.

Debugging Accelerator Type

The debugging accelerator type to use for debugging the GPU code. (Available when the GPU debugger is active.)

GPU Default Breakpoint Behavior

Sets how often the GPU debugger breaks.

Choices

- **Break once per warp** - Break once per warp
- **Break for every thread (like CPU behavior)** - Break for every thread (like CPU behavior)

Attach

Specifies whether the debugger should attempt to attach to an existing process when debugging starts.

SQL Debugging

Attach the SQL debugger.

Deployment Directory

When debugging on a remote machine, if you want the contents of the project output (except for PDB files) to be copied to the remote machine, specify the path here.

Additional Files to Deploy

When debugging on a remote machine, files and directories specified here (besides the project output) are copied to the Deployment Directory if one was specified.

Deploy Visual C++ Debug Runtime Libraries

Specifies whether to deploy the debug runtime libraries for the active platform (Win32, x64, or ARM).

Amp Default Accelerator

Override C++ AMP's default accelerator selection. Property does not apply when debugging managed code.

Web Browser Debugger Property Page

HTTP URL

Specifies the URL for the project.

Debugger Type

Specifies the debugger type to use. When set to Auto, the debugger type will be selected based on contents of the exe file.

Choices

- **Native Only** - Native Only
- **Managed Only** - Managed Only
- **Mixed** - Mixed
- **Auto** - Auto
- **Script** - Script

Web Service Debugger Property Page

HTTP URL

Specifies the URL for the project.

Debugger Type

Specifies the debugger type to use. When set to Auto, the debugger type will be selected based on contents of the exe file.

Choices

- **Native Only** - Native Only
- **Managed Only** - Managed Only
- **Mixed** - Mixed

- **Auto** - Auto
- **Script** - Script

SQL Debugging

Attach the SQL debugger.

VC++ Directories Property Page (Windows)

Article • 04/10/2023

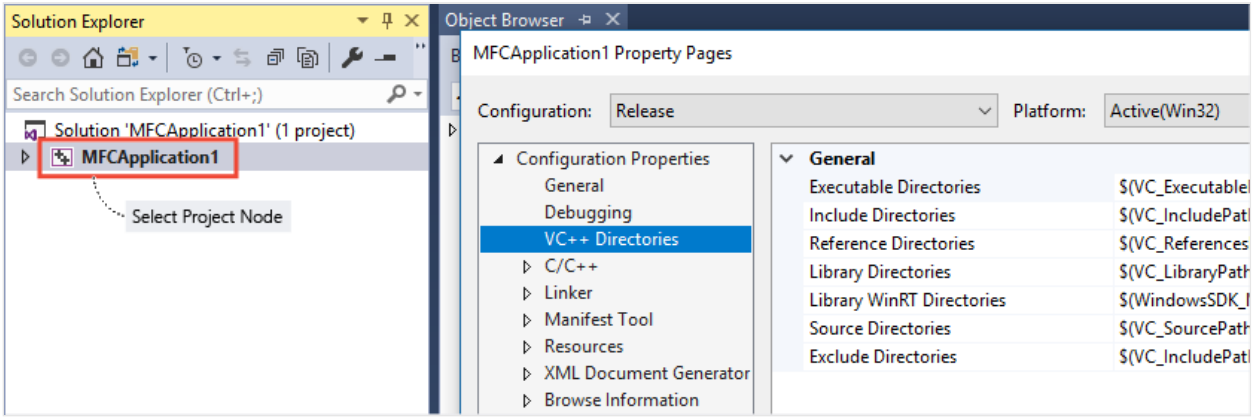
Use this property page to tell Visual Studio which directories to use when building the currently selected project. To set directories for multiple projects in a solution, use a custom property sheet as described in Share or reuse Visual Studio C++ project settings.

For the Linux version of this page, see VC++ Directories (Linux C++).

To access the **VC++ Directories** property page:

1. If the **Solution Explorer** window isn't visible, choose **View** > **Solution Explorer** on the main menu.
2. Right-click on a project node (not the top-level solution) and choose **Properties** to open the **Property Pages** dialog box.
3. Select the **Configuration Properties** > **VC++ Directories** property page.

VC++ Directories properties apply to a project, not the top-level solution node. If you don't see **VC++ Directories** under **Configuration Properties**, select a C++ project node in the **Solution Explorer** window:



The **VC++ Directories** property page for cross-platform projects looks different. For information specific to Linux C++ projects, see VC++ Directories (Linux C++).

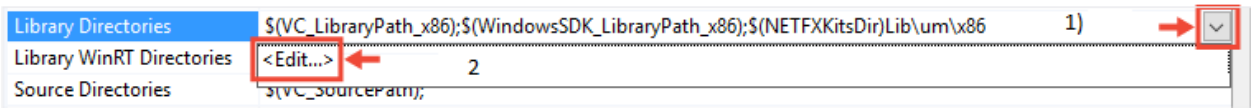
If you're not familiar with *project properties* in Visual Studio, you might find it helpful to first read Set C++ compiler and build properties in Visual Studio.

The default settings for **VC++ Directories** properties depend on project type. For desktop projects, they include the C++ tools locations for a particular Platform Toolset

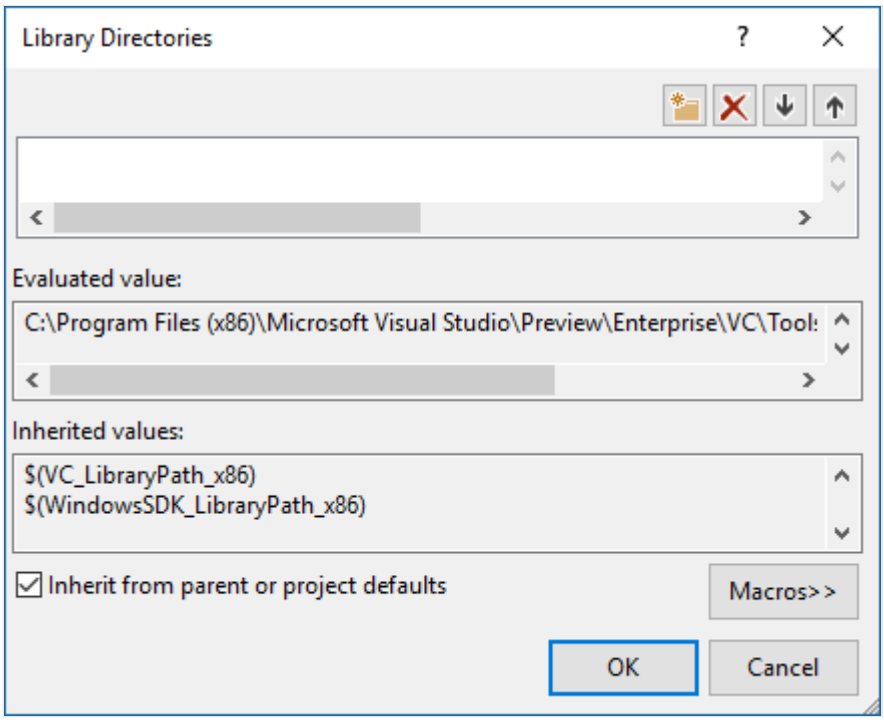
and the Windows SDK location. You can change the **Platform Toolset** and **Windows SDK version** on the **Configuration Properties** > **General** page.

To view the values for any of the directories:

1. Select one of the properties in the **VC++ Directories** page. For example, choose **Library Directories**.
2. Choose the down-arrow button at the end of the property value field.
3. In the drop-down menu, choose **Edit**.



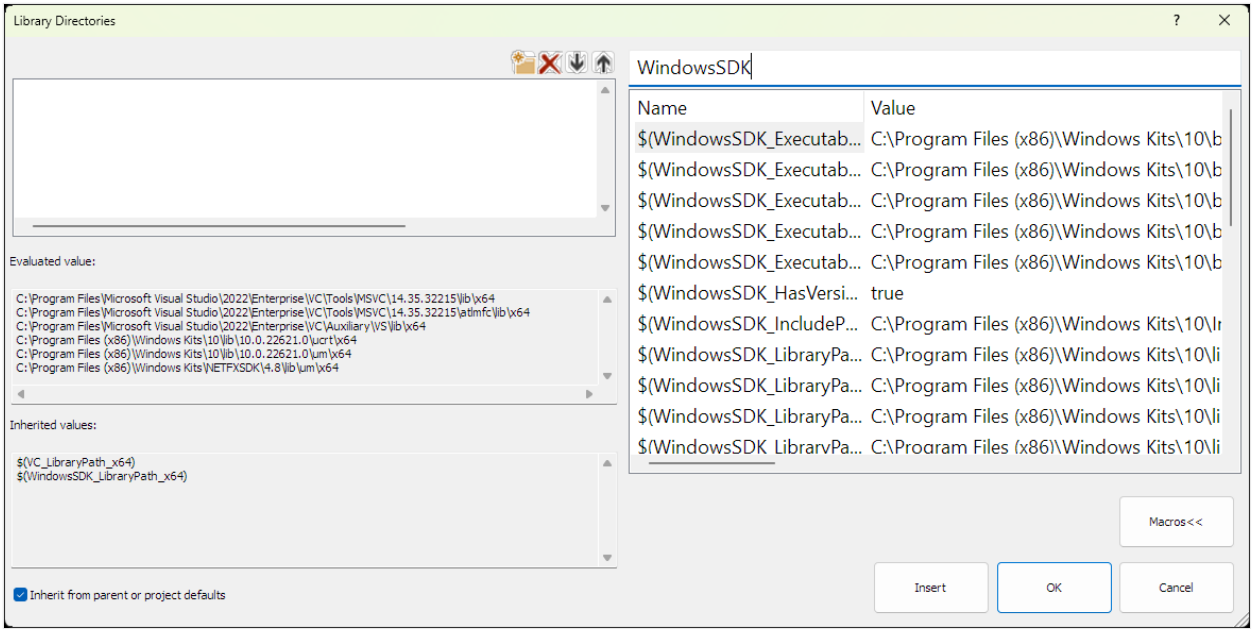
You now see a dialog box like this:



Use this dialog to view the current directories. However, if you want to change or add a directory, it's better to use **Property Manager** to create a property sheet or modify the default user property sheet. For more information, see Share or reuse Visual Studio C++ project settings.

As shown earlier, many of the inherited paths are provided as macros. To examine the current value of a macro, choose the **Macros** button in the lower right corner of the dialog box. Many macros depend on the configuration type. A macro in a debug build might evaluate to a different path than the same macro in a release build, for example. For information about examining macros values, see Common macros for build commands and properties.

You can search for partial or complete matches of a macro in the edit box. The following screenshot shows all the macros that contain the string "WindowsSDK". It also shows the current path that each macro evaluates to:



This list is populated as you type. Don't press **Enter**.

For more information about macros and why you should use them instead of hard-coded paths whenever possible, see Set C++ compiler and build properties in Visual Studio.

For information about examining the values of the macros, see Common macros for build commands and properties. That topic also lists commonly used macros.

You can define your own macros in two ways:

- Set environment variables in a developer command prompt. All environment variables are treated as MSBuild properties/macros.
- Define user macros in a `.props` file. For more information, see Property page macros.

For more information, see Property inheritance in Visual Studio projects, and these blog posts: VC++ Directories, Visual Studio 2010 C++ Project Upgrade Guide.

General

You can also specify other directories, as follows.

Executable Directories
Directories in which to search for executable files. Corresponds to the `PATH` environment

variable.

Include Directories
Directories in which to search for include files that are referenced in the source code. Corresponds to the `INCLUDE` environment variable.

External Include Directories
Paths for include files to treat as external or system files during compilation. These files are skipped in build up-to-date checks. These paths are also used by the External Includes properties. For more information on how to set these options in the IDE, see the /external compiler option.

Reference Directories
Directories in which to search for assembly and module (metadata) files that are referenced in the source code by the #using directive. Corresponds to the `LIBPATH` environment variable.

Library Directories
Directories to search for library (*`.lib`*) files. This search includes run-time libraries. Corresponds to the `LIB` environment variable. This setting doesn't apply to *`.obj`* files; to link to an *`.obj`* file, on the **Configuration Properties** > **Linker** > **General** property page, select **Additional Library Dependencies** and then specify the relative path of the file. For more information, see Linker property pages.

Library WinRT Directories
Directories to search for WinRT library files for use in Universal Windows Platform (UWP) apps.

Source Directories
Directories in which to search for source files to use for IntelliSense.

Exclude Directories
Before each compilation, Visual Studio queries the timestamp on all files to determine whether any have been modified since the previous compilation. If your project has large stable libraries, you can potentially speed up build times by excluding those directories from the timestamp check.

Public Project Content

Public Include Directories
One or more directories to automatically add to the include path in projects that reference this project.

All Header Files are Public
Specifies whether to automatically add public directories or all project header files to the include path in projects that reference this project.

Public C++ Module Directories
One or more directories that contain C++ module or header unit sources to make available automatically to projects that reference this project.

All Modules are Public
Specifies whether to make all project modules and header units available automatically to projects that reference this project.

Sharing the settings

You can share project properties with other users or across multiple computers. For more information, see Set C++ compiler and build properties in Visual Studio.

C/C++ Property Pages

Article • 06/09/2023

The following property pages are found under **Project** > **Properties** > **Configuration Properties** > **C/C++**:

C/C++ General Properties

Additional Include Directories

Specifies one or more directories to add to the include path. Separate directories with semi-colons ('`;`') if there's more than one. Sets the /I (Additional include directories) compiler option.

Additional #using Directories

Specifies one or more directories to search to resolve names passed to a `#using` directive. Separate directories with semi-colons ('`;`') if there's more than one. Sets the /AI compiler option.

Additional BMI Directories

Specifies one or more directories to search to resolve names passed to an `import` directive. Separate directories with semi-colons ('`;`') if there's more than one. Sets the `/ifcSearchDir[path]` compiler option.

Additional Module Dependencies

Specifies one or more modules to use to resolve names passed to an `import` directive. Separate directories with semi-colons ('`;`') if there's more than one. Sets the /reference compiler option.

Additional Header Unit Dependencies

Specifies one or more header units to use to resolve names passed to an `import` header directive. Separate directories with semi-colons ('`;`') if there's more than one. Sets the /headerUnit compiler option.

Scan Sources for Module Dependencies

When set to **Yes**, the compiler scans all C++ sources, not just module interface and header unit sources, for module and header units dependencies. The build system builds the full dependencies graph, which ensures that all imported modules and header units are built before compiling the files that depend on them. When combined with **Translate Includes to Imports**, any header file that's specified in a header-units.json file in the same directory as the header file is compiled into a header unit.

Files that have the extension `.ixx`, and files that have their **File properties** > **C/C++** > **Compile As** property set to **Compile as C++ Header Unit (/exportHeader)**, are always scanned.

Translate Includes to Imports

When set to **Yes**, the compiler treats a `#include` directive as an `import` directive if certain conditions are met: The header file is specified in a `header-units.json` file in the same directory, and a compiled header unit (an `.ifc` file) is available for the header file. Otherwise, the header file is treated as a normal `#include`. The `header-units.json` file is used to build header units for each `#include` without symbol duplication. When combined with **Scan Sources for Module Dependencies**, the compiler automatically finds all of the header files that can be compiled into header units. This property sets the /translateInclude compiler option.

Debug Information Format

Specifies the type of debugging information generated by the compiler. This property requires compatible linker settings. Sets /Z7, /Zi, /ZI (Debug information format) compiler options.

Choices

- **None** - Produces no debugging information, so compilation may be faster.
- **C7 compatible** - Select the type of debugging information created for your program and whether this information is kept in object (.obj) files or in a program database (PDB).
- **Program Database** - Produces a program database (PDB) that contains type information and symbolic debugging information for use with the debugger. The symbolic debugging information includes the names and types of variables and functions, and line numbers.

- **Program Database for Edit And Continue** - Produces a program database, as described previously, in a format that supports the Edit and Continue feature.

Support Just My Code Debugging

Adds supporting code for enabling Just My Code debugging in this compilation unit. Sets /JMC.

Common Language RunTime Support

Use the .NET runtime service. This switch is incompatible with some other switches; see the documentation on the /clr family of switches for details.

Choices

- **No Common Language RunTime Support** - No Common Language RunTime Support
- **Common Language RunTime Support** - Creates metadata for your application that can be consumed by other CLR applications. Also allows your application to consume types and data in the metadata of other CLR components.
- **Pure MSIL Common Language RunTime Support** - Produces an MSIL-only output file with no native executable code, although it can contain native types compiled to MSIL.
- **Safe MSIL Common Language RunTime Support** - Produces an MSIL-only (no native executable code) and verifiable output file.

Consume Windows Runtime Extension

Consume the Windows Run Time languages extensions. Sets /ZW.

Suppress Startup Banner

Suppresses the display of the sign-on banner when the compiler starts up and display of informational messages during compiling.

Warning Level

Select how strict you want the compiler to be about code errors. Sets /W0 - /W4.

Choices

- **Turn Off All Warnings** - Level 0 disables all warnings.
- **Level1** - Level 1 displays severe warnings. Level 1 is the default warning level at the command line.
- **Level2** - Level 2 displays all level 1 warnings and warnings less severe than level 1.
- **Level3** - Level 3 displays all level 2 warnings and all other warnings recommended for production purposes.
- **Level4** - Level 4 displays all level 3 warnings plus informational warnings, which in most cases can be safely ignored.
- **EnableAllWarnings** - Enables all warnings, including the ones disabled by default.

Treat Warnings As Errors

Treats compiler warnings as errors. For a new project, it may be best to use /WX in every compilation. Resolve all warnings to minimize hard-to-find code defects.

Warning Version

Hide warnings introduced after a specific version of the compiler. Sets /Wv:xx[.yy[.zzzzz]].

Diagnostics Format

Enables rich diagnostics, with column information and source context in diagnostic messages.

Choices

- **Caret** - Provides column information in the diagnostic message. And, outputs the relevant line of source code with a caret that indicates the offending column.
- **Column Info** - Additionally provides the column number within the line where the diagnostic is issued, where applicable.
- **Classic** - Outputs only the prior, concise diagnostic messages with the line number.

SDL checks

Additional Security Development Lifecycle (SDL) recommended checks; includes enabling additional secure code generation features and enables extra security-relevant warnings as errors. Sets /sdl, /sdl-.

Multi-processor Compilation

Enable multi-processor compilation. Sets the /MP compiler option.

Enable Address Sanitizer

Compiles and links the program with AddressSanitizer instrumentation. This property currently supports x86 and x64 target builds. Sets the /fsanitize compiler option.

C/C++ Optimization Properties

Optimization

Select option for code optimization; choose Custom to use specific optimization options. Sets /Od, /O1, /O2.

Choices

- **Custom** - Custom optimization.
- **Disabled** - Disable optimization.
- **Maximum Optimization (Favor Size)** - Equivalent to `/Os /Oy /Ob2 /Gs /GF /Gy`
- **Maximum Optimization (Favor Speed)** - Equivalent to `/Oi /Ot /Oy /Ob2 /Gs /GF /Gy`
- **Optimizations (Favor Speed)** - Equivalent to `/Oi /Ot /Oy /Ob2`

Inline Function Expansion

Select the level of inline function expansion for the build. Sets /Ob.

Choices

- **Default**
- **Disabled** - Disables inline expansion, which is on by default.
- **Only __inline** - Expands only functions marked as `inline`, `__forceinline`, or `__inline`. Or, in a C++ member function, defined within a class declaration.
- **Any Suitable** - Expands functions marked as `inline` or `__inline` and any other function that the compiler chooses. (Expansion occurs at the compiler's discretion, often referred to as *autoinlining*.)

Enable Intrinsic Functions

Enables intrinsic functions. Using intrinsic functions generates faster, but possibly larger, code. Sets /Oi.

Favor Size Or Speed

Whether to favor code size or code speed; 'Global Optimization' must be turned on. Sets /Ot, /Os.

Choices

- **Favor small code** - Minimizes the size of EXEs and DLLs by instructing the compiler to favor size over speed.
- **Favor fast code** - Maximizes the speed of EXEs and DLLs by instructing the compiler to favor speed over size. (This value is the default.)
- **Neither** - No size and speed optimization.

Omit Frame Pointers

Suppresses creation of frame pointers on the call stack.

Enable Fiber-Safe Optimizations

Enables memory space optimization when using fibers and thread local storage access. Sets /GT.

Whole Program Optimization

Enables cross-module optimizations by delaying code generation to link time. Requires the linker option **Link Time Code Generation**. Sets /GL.

C/C++ Preprocessor Properties

Preprocessor Definitions

Defines preprocessing symbols for your source file.

Undefine Preprocessor Definitions

Specifies one or more preprocessor undefines. Sets /U.

Undefine All Preprocessor Definitions

Undefine all previously defined preprocessor values. Sets /u.

Ignore Standard Include Paths

Prevents the compiler from searching for include files in directories specified in the INCLUDE environment variables.

Preprocess to a File

Preprocesses C and C++ source files, and writes the preprocessed output to a file. This option suppresses compilation, and it doesn't produce an `.obj` file.

Preprocess Suppress Line Numbers

Preprocess without #line directives.

Keep Comments

Suppresses comment strip from source code; requires setting at least one of the **Preprocessing** options. Sets /C.

C/C++ Code Generation Properties

Enable String Pooling

The compiler creates only one read-only copy of identical strings in the program image. It results in smaller programs, an optimization called *string pooling*. /O1, /O2, and /ZI automatically set /GF option.

Enable Minimal Rebuild

Enables minimal rebuild, which determines whether to recompile C++ source files that include changed C++ class definitions, stored in header `.h` files.

Enable C++ Exceptions

Specifies the model of exception handling to be used by the compiler.

Choices

- **Yes with SEH Exceptions** - The exception-handling model that catches asynchronous (structured) and synchronous (C++) exceptions. Sets /EHa.
- **Yes** - The exception-handling model that catches C++ exceptions only and tells the compiler to assume that extern C functions never throw a C++ exception. Sets /EHsc.
- **Yes with Extern C functions** - The exception-handling model that catches C++ exceptions only and tells the compiler to assume that extern C functions do throw an exception. Sets /EHs.
- **No** - No exception handling.

Smaller Type Check

Enable checking for conversion to smaller types, incompatible with any optimization type other than debug. Sets /RTCc.

Basic Runtime Checks

Enable basic runtime error checks, incompatible with any optimization type other than debug. Sets /RTCs, /RTCu, /RTC1.

Choices

- **Stack Frames** - Enables stack frame run-time error checking.
- **Uninitialized variables** - Reports when a variable is used without having been initialized.
- **Both (/RTC1, equiv. to /RTCsu)** - Equivalent of `/RTCsu`.
- **Default** - Default runtime checks.

Runtime Library

Specify runtime library for linking. Sets /MT, /MTd, /MD, /MDd.

Choices

- **Multi-threaded** - Causes your application to use the multithread, static version of the run-time library.
- **Multi-threaded Debug** - Defines `_DEBUG` and `_MT`. This option also causes the compiler to place the library name `LIBCMTD.lib` into the `.obj` file so that the linker

will use `LIBCMTD.lib` to resolve external symbols.

- **Multi-threaded DLL** - Causes your application to use the multithread- and DLL-specific version of the run-time library. Defines `_MT` and `_DLL` and causes the compiler to place the library name *MSVCRT.lib* into the `.obj` file.
- **Multi-threaded Debug DLL** - Defines `_DEBUG`, `_MT`, and `_DLL` and causes your application to use the debug multithread- and DLL-specific version of the run-time library. It also causes the compiler to place the library name `MSVCRTD.lib` into the `.obj` file.

Struct Member Alignment

Specifies 1, 2, 4, or 8-byte boundaries for struct member alignment. Sets /Zp.

Choices

- **1 Byte** - Packs structures on one-byte boundaries. Same as `/Zp`.
- **2 Bytes** - Packs structures on two-byte boundaries.
- **4 Bytes** - Packs structures on four-byte boundaries.
- **8 Bytes** - Packs structures on eight-byte boundaries (default).
- **16 Bytes** - Packs structures on sixteen-byte boundaries.
- **Default** - Default alignment settings.

Security Check

The Security Check helps detect stack-buffer over-runs, a common attempted attack upon a program's security.

Choices

- **Disable Security Check** - Disable Security Check. Sets /GS-.
- **Enable Security Check** - Enable Security Check. Sets /GS.

Control Flow Guard

Guard security check helps detect attempts to dispatch to illegal block of code.

Choices

- **Yes** - Enable Security Check with Guard Sets /guard:cf.

- No

Enable Function-Level Linking

Allows the compiler to package individual functions in the form of packaged functions (COMDATs). Required for edit and continue to work. Sets /Gy.

Enable Parallel Code Generation

Allows the compiler to generate parallel code for loops identified using `#pragma loop(hint_parallel[(n)])` when optimization is enabled.

Enable Enhanced Instruction Set

Enable use of instructions found on processors that support enhanced instruction sets. For example, the SSE, SSE2, AVX, and AVX2 enhancements to IA-32. And, the AVX and AVX2 enhancements to x64. Currently **`/arch:SSE`** and **`/arch:SSE2`** are only available when building for the x86 architecture. If no option is specified, the compiler uses instructions found on processors that support SSE2. Use of enhanced instructions can be disabled with **`/arch:IA32`**. For more information, see /arch (x86), /arch (x64), /arch (ARM64), and /arch (ARM).

Choices

- **Streaming SIMD Extensions** - Streaming SIMD Extensions. Sets **`/arch:SSE`**
- **Streaming SIMD Extensions 2** - Streaming SIMD Extensions 2. Sets **`/arch:SSE2`**
- **Advanced Vector Extensions** - Advanced Vector Extensions. Sets **`/arch:AVX`**
- **Advanced Vector Extensions 2** - Advanced Vector Extensions 2. Sets **`/arch:AVX2`**
- **No Enhanced Instructions** - No Enhanced Instructions. Sets **`/arch:IA32`**
- **Not Set** - Not Set.

Floating Point Model

Sets the floating point model. Sets /fp:precise, /fp:strict, /fp:fast.

Choices

- **Precise** - Default. Improves the consistency of floating-point tests for equality and inequality.

- **Strict** - The strictest floating-point model. `/fp:strict` causes `fp_contract` to be OFF and `fenv_access` to be ON. `/fp:except` is implied and can be disabled by explicitly specifying `/fp:except-`. When used with `/fp:except-`, `/fp:strict` enforces strict floating-point semantics but without respect for exceptional events.
- **Fast** - Creates the fastest code in most cases.

Enable Floating Point Exceptions

Reliable floating-point exception model. Exceptions will be raised immediately after they're triggered. Sets /fp:except.

Create Hotpatchable Image

When hotpatching is on, the compiler ensures that first instruction of each function is two bytes, as required for hot patching. Sets /hotpatch.

Spectre Mitigation

Spectre mitigations for CVE 2017-5753. Sets /Qspectre.

Choices

- **Enabled** - Enable Spectre mitigation feature for CVE 2017-5753
- **Disabled** - Not Set.

C/C++ Language Properties

Disable Language Extensions

Suppresses or enables language extensions. Sets /Za.

Treat WChar_t As Built in Type

When specified, the type `wchar_t` becomes a native type that maps to `__wchar_t` in the same way that `short` maps to `__int16`. /Zc:wchar_t is on by default.

Force Conformance in For Loop Scope

Implements standard C++ behavior for the `for` statement loops with Microsoft extensions. Sets /Za, /Ze (Disable language extensions. /Zc:forScope is on by default.

Remove unreferenced code and data

When specified, the compiler no longer generates symbol information for unreferenced code and data.

Enforce type conversion rules

Used to identify an rvalue reference type as the result of a cast operation according to the C++11 standard.

Enable Run-Time Type Information

Adds code for checking C++ object types at run time (*runtime type information*, or RTTI). Sets /GR, /GR-.

Open MP Support

Enables OpenMP 2.0 language extensions. Sets /openmp.

C++ Language Standard

Determines the C++ language standard that the compiler enables. The default value doesn't set a standard option, so the compiler uses its default C++14 setting. If you select a specific value, the corresponding /std compiler option is set.md).

Choices

- **Default (ISO C++14 Standard)**
- **ISO C++14 Standard (/std:c++14)**
- **ISO C++17 Standard (/std:c++17)**
- **ISO C++20 Standard (/std:c++20)**
- **Preview - Features from the Latest C++ Working Draft (/std:c++latest)**

C Language Standard

Determines the C language standard that the compiler enables. The default value doesn't set a standard option, so the compiler uses its default legacy MSVC setting. If

you select a specific value, the corresponding /std compiler option is set.md).

Choices

- **Default (Legacy MSVC)**
- **ISO C11 Standard (/std:c11)**
- **ISO C17 (2018) Standard (/std:c17)**

Conformance mode

Enables or suppresses conformance mode. Sets /permissive-.

Enable Experimental C++ Standard Library Modules

Experimental support for the C++ Modules TS and Standard Library modules.

Build ISO C++23 Standard Library Modules

Starting in Visual Studio 17.6, when this property is enabled and C++ Language Standard is set to `/std:c++latest`, Visual C++ projects automatically find and build ISO C++23 Standard Library modules. This enables you to `import std` or `import std.compat` in your C++ code.

C/C++ Precompiled Headers Properties

Create/Use Precompiled Header

Enables creation or use of a precompiled header during the build. Sets /Yc, /Yu.

Choices

- **Create** - Instructs the compiler to create a precompiled header (`.pch`) file that represents the state of compilation at a certain point.
- **Use** - Instructs the compiler to use an existing precompiled header (`.pch`) file in the current compilation.
- **Not Using Precompiled Headers** - Not using precompiled headers.

Precompiled Header File

Specifies header file name to use when creating or using a precompiled header file. Sets /Yc, /Yu.

Precompiled Header Output File

Specifies the path or name of the generated precompiled header file. Sets /Fp.

C/C++ Output Files Properties

Expand Attributed Source

Create listing file with expanded attributes injected into source file. Sets /Fx.

Assembler Output

Specifies the contents of assembly language output file. Sets /FA, /FAc, /FAs, /FAcs.

Choices

- **No Listing** - No listing.
- **Assembly-Only Listing** - Assembly code; `.asm`
- **Assembly With Machine Code** - Machine and assembly code; `.cod`
- **Assembly With Source Code** - Source and assembly code; `.asm`
- **Assembly, Machine Code and Source** - Assembly, machine code and source code; `.cod`

Use Unicode For Assembler Listing

Causes the output file to be created in UTF-8 format.

ASM List Location

Specifies relative path or name for ASM listing file; can be file or directory name. Sets /Fa.

Object File Name

Specifies a name to override the default object file name; can be file or directory name. Sets /Fo.

Program Database File Name

Specifies a name for a compiler-generated PDB file; also specifies base name for the required compiler-generated IDB file; can be file or directory name. Sets /Fd.

Generate XML Documentation Files

Specifies that the compiler should generate XML documentation comment files (.XDC). Sets /doc.

XML Documentation File Name

Specifies the name of the generated XML documentation files; can be file or directory name. Sets /doc:<name>.

C/C++ Browse Information Properties

Enable Browse Information

Specifies level of browse information in `.bsc` file. Sets /FR.

Browse Information File

Specifies optional name for browser information file. Sets /FR<name>.

External Includes

Treat Files Included with Angle Brackets as External

Specifies whether to treat files included with angle brackets as external. Set this property to **Yes** to set the /external:anglebrackets compiler option.

External Header Warning Level

Select how strict you want the compiler to be about code errors in external headers. This property sets the /external:Wn compiler option. If this value is set to **Inherit Project Warning Level** or the default, other `/external` options are ignored.

Template Diagnostics in External Headers

Specifies whether to evaluate the warning level across a template instantiation chain. Set this property to **Yes** to set the /external:templates- compiler option.

Disable Code Analysis for External Headers

Disables code analysis for external headers. Sets the /analyze:external- compiler option.

Analysis Ruleset for External Headers

Specifies a code analysis ruleset override for external headers. If not specified, the Code Analysis setting is used. Sets the /analyze:external:ruleset path compiler option.

C/C++ Advanced Properties

Calling Convention

Select the default calling convention for your application (can be overridden by function). Sets /Gd, /Gr, /Gz, /Gv.

Choices

- `__cdecl` - Specifies the `__cdecl` calling convention for all functions except C++ member functions and functions marked `__stdcall` or `__fastcall`.
- `__fastcall` - Specifies the `__fastcall` calling convention for all functions except C++ member functions and functions marked `__cdecl` or `__stdcall`. All `__fastcall` functions must have prototypes.
- `__stdcall` - Specifies the `__stdcall` calling convention for all functions except C++ member functions and functions marked `__cdecl` or `__fastcall`. All `__stdcall` functions must have prototypes.
- `__vectorcall` - Specifies the `__vectorcall` calling convention for all functions except C++ member functions and functions marked `__cdecl`, `__fastcall`, or `__stdcall`. All `__vectorcall` functions must have prototypes.

Compile As

Select compile language option for source files. Sets /TC, /TP, /interface, /internalPartition, or /exportHeader options.

Choices

- **Default** - Default.
- **Compile as C Code (/TC)** - Compile specified source files as C code. By default, files with a `.c` extension are compiled as C.
- **Compile as C++ Code (/TP)** - Compile specified source files as C++ code. By default, all source files that don't have a `.c`, `.ixx`, `.cppm`, `.h`, or no extension are compiled as C++.
- **Compile as C++ Module Code (/interface)** - Compile specified source files as C++ module code. By default, files with a `.ixx` or `.cppm` extension are compiled as C++ module code.
- **Compile as C++ Module Internal Partition (/internalPartition)** - Compile specified source files as C++ module internal partition.
- **Compile as C++ Header Unit (/exportHeader)** - Compile specified source files as C++ header unit. By default, files with a `.h` extension or no extension are compiled as header units.

Disable Specific Warnings

Disable the specified warning numbers. Put the warning numbers in a semi-colon delimited list. Sets /wd<number>.

Forced Include File

one or more forced include files. Sets /FI<name>.

Forced #using File

Specifies one or more forced #using files. Sets /FU<name>.

Show Includes

Generates a list of include files with compiler output. Sets /showIncludes.

Use Full Paths

Use full paths in diagnostic messages. Sets /FC.

Omit Default Library Name

Doesn't include default library names in `.obj` files. Sets /Zl.

Internal Compiler Error Reporting

> **Note**
>
> This option is deprecated. Starting in Windows Vista, error reporting is controlled by **Windows Error Reporting (WER)** settings.

Treat Specific Warnings As Errors

Treats the specific compiler warning as an error where n is a compiler warning.

Additional Options

Additional Options.

Linker Property Pages

Article • 09/22/2022

The following properties are found under **Project** > **Properties** > **Configuration Properties** > **Linker**. For more information about the linker, see CL Invokes the Linker and Linker Options.

General Property Page

Output File

The /OUT option overrides the default name and location of the program that the linker creates.

Show Progress

Prints Linker Progress Messages

Choices

- **Not Set** - No verbosity.
- **Display all progress messages** - Displays all progress messages.
- **For Libraries Searched** - Displays progress messages indicating just the libraries searched.
- **About COMDAT folding during optimized linking** - Displays information about COMDAT folding during optimized linking.
- **About data removed during optimized linking** - Displays information about functions and data removed during optimized linking.
- **About Modules incompatible with SEH** - Displays information about modules incompatible with Safe Exception Handling.
- **About linker activity related to managed code** - Display information about linker activity related to managed code.

Version

The /VERSION option tells the linker to put a version number in the header of the `.exe` or `.dll` file. Use `DUMPBIN /HEADERS` to see the image version field of the `OPTIONAL HEADER VALUES` to see the effect of `/VERSION`.

Enable Incremental Linking

Enables incremental linking. (/INCREMENTAL, /INCREMENTAL:NO)

Suppress Startup Banner

The /NOLOGO option prevents display of the copyright message and version number.

Ignore Import Library

This property tells the linker not to link any `.lib` output generated from this build into any dependent project. It allows the project system to handle `.dll` files that don't produce a `.lib` file when built. If a project depends on another project that produces a DLL, the project system automatically links the `.lib` file produced by that child project. This property may be unnecessary in projects that produce COM DLLs or resource-only DLLs, because these DLLs don't have any meaningful exports. If a DLL has no exports, the linker doesn't generate a `.lib` file. If no export `.lib` file is present, and the project system tells the linker to link with the missing DLL, the link fails. Use the **Ignore Import Library** property to resolve this problem. When set to **Yes**, the project system ignores the presence or absence of the `.lib` file, and causes any project that depends on this project to not link with the nonexistent `.lib` file.

To programmatically access this property, see IgnoreImportLibrary.

Register Output

Runs `regsvr32.exe /s $(TargetPath)` on the build output, which is valid only on `.dll` projects. For `.exe` projects, this property is ignored. To register an `.exe` output, set a postbuild event on the configuration to do the custom registration that is always required for registered `.exe` files.

To programmatically access this property, see RegisterOutput.

Per-user Redirection

Registration in Visual Studio has traditionally been done in `HKEY_CLASSES_ROOT` (HKCR). With Windows Vista and later operating systems, to access HKCR you must run Visual Studio in elevated mode. Developers don't always want to run in elevated mode but still must work with registration. Per-user redirection allows you to register without having to run in elevated mode.

Per-user redirection forces any writes to HKCR to be redirected to `HKEY_CURRENT_USER` (HKCU). If per-user redirection is turned off, it can cause Project Build Error PRJ0050 when the program tries to write to HKCR.

Additional Library Directories

Allows the user to override the environment's library path. (/LIBPATH:folder)

Link Library Dependencies

Specifies whether to link the *`.lib`* files that are produced by dependent projects. Typically, you want to link in the *`.lib`* files, but it may not be the case for certain DLLs.

You can also specify a *`.obj`* file by providing the file name and relative path, for example, *`..\..\MyLibProject\MyObjFile.obj`*. If the source code for the *`.obj`* file has a `#include` for a precompiled header, for example, *`pch.h`*, then the *`pch.obj`* file is located in the same folder as *`MyObjFile.obj`*. You must also add *`pch.obj`* as an additional dependency.

Use Library Dependency Inputs

Specifies whether to use the inputs to the librarian tool, rather than the library file itself, when linking in library outputs of project dependencies. In a large project, when a dependent project produces a *`.lib`* file, incremental linking is disabled. If there are many dependent projects that produce *`.lib`* files, building the application can take a long time. When this property is set to **Yes**, the project system links in the *`.obj`* files for *`.lib`* files produced by dependent projects, enabling incremental linking.

For information about how to access the **General** linker property page, see Set compiler and build properties.

Link Status

Specifies whether the linker should display a progress indicator showing what percentage of the link is complete. The default is to not display this status information. (/LTCG:STATUS|LTCG:NOSTATUS)

Prevent DLL Binding

/ALLOWBIND:NO sets a bit in a DLL's header that indicates to `Bind.exe` that binding the image isn't allowed. You may not want a DLL to be bound if it has been digitally signed (binding invalidates the signature).

Treat Linker Warning As Errors

/WX causes no output file to be generated if the linker generates a warning.

Force File Output

The /FORCE option tells the linker to create an `.exe` file or DLL even if a symbol is referenced but not defined (`UNRESOLVED`), or is defined multiple times (`MULTIPLE`). It may create an invalid `.exe` file.

Choices

- **Enabled** - `/FORCE` with no arguments implies both `/FORCE:MULTIPLE` and `/FORCE:UNRESOLVED`.
- **Multiply Defined Symbol Only** - Use `/FORCE:MULTIPLE` to create an output file, even if LINK finds more than one definition for a symbol.
- **Undefined Symbol Only** - Use `/FORCE:UNRESOLVED` to create an output file whether or not LINK finds an undefined symbol. `/FORCE:UNRESOLVED` is ignored if the entry point symbol is unresolved.

Create Hot Patchable Image

Prepares an image for hot patching.

Choices

- **Enabled** - Prepares an image for hot patching.
- **X86 Image Only** - Prepares an X86 image for hot patching.
- **X64 Image Only** - Prepares an X64 image for hot patching.
- **Itanium Image Only** - Prepares an Itanium image for hot patching.

Specify Section Attributes

The /SECTION option changes the attributes of a section, overriding the attributes set when the `.obj` file for the section was compiled.

Input Property Page

Additional Dependencies

Specifies extra dependency items to add to the link command line, for example `kernel32.lib`.

Ignore All Default Libraries

The /NODEFAULTLIB option tells the linker to remove one or more default libraries from the list of libraries it searches when resolving external references.

Ignore Specific Default Libraries

Specifies one or more names of default libraries to ignore. Separate multiple libraries with semi-colons. (/NODEFAULTLIB:[name, name, ...])

Module Definition File

The /DEF option passes a module-definition file (`.def`) to the linker. Only one `.def` file can be specified to LINK.

Add Module to Assembly

The /ASSEMBLYMODULE option allows you to add a module reference to an assembly. Type information in the module won't be available to the assembly program that added the module reference. However, type information in the module will be available to any program that references the assembly.

Embed Managed Resource File

/ASSEMBLYRESOURCE embeds a resource file in the output file.

Force Symbol References

The /INCLUDE option tells the linker to add a specified symbol to the symbol table.

Delay Loaded DLLs

The /DELAYLOAD option causes delayed loading of DLLs. The dll name specifies a DLL to delay load.

Assembly Link Resource

The /ASSEMBLYLINKRESOURCE option creates a link to a .NET Framework resource in the output file. The linker doesn't place the resource file in the output file.

Manifest File Property Page

Generate Manifest

/MANIFEST specifies that the linker should create a side-by-side manifest file.

Manifest File

/MANIFESTFILE lets you change the default name of the manifest file. The default name of the manifest file is the file name with `.manifest` appended.

Additional Manifest Dependencies

/MANIFESTDEPENDENCY lets you specify attributes that will be placed in the dependency section of the manifest file.

Allow Isolation

Specifies behavior for manifest lookup. (/ALLOWISOLATION:NO)

Enable User Account Control (UAC)

Specifies whether or not User Account Control is enabled. (/MANIFESTUAC, /MANIFESTUAC:NO)

UAC Execution Level

Specifies the requested execution level for the application when running with User Account Control. (/MANIFESTUAC:level=[value])

Choices

- **asInvoker** - UAC Execution Level: as invoker.
- **highestAvailable** - UAC Execution Level: highest available.
- **requireAdministrator** - UAC Execution Level: require administrator.

UAC Bypass UI Protection

Specifies whether or not to bypass user interface protection levels for other windows on the desktop. Set this property to 'Yes' only for accessibility applications. (/MANIFESTUAC:uiAccess=[true | false])

Debugging Property Page

Generate Debug Info

This option enables creation of debugging information for the `.exe` file or the DLL.

Choices

- **No** - Produces no debugging information.
- **Generate Debug Information** - Create a complete Program Database (PDB) ideal for distribution to Microsoft Symbol Server.
- **Generate Debug Information optimized for faster links** - Produces a program database (PDB) ideal for a fast edit-link-debug cycle.
- **Generate Debug Information optimized for sharing and publishing** - Produces a program database (PDB) ideal for a shared edit-link-debug cycle.

Generate Program Database File

By default, when /DEBUG is specified, the linker creates a program database (PDB) which holds debugging information. The default file name for the PDB has the base name of the program and the extension `.pdb`.

Strip Private Symbols

The /PDBSTRIPPED option creates a second program database (PDB) file when you build your program image with any of the compiler or linker options that generate a PDB file (`/DEBUG`, `/Z7`, `/Zd`, or `/Zi`).

Generate Map File

The /MAP option tells the linker to create a mapfile.

Map File Name

A user-specified name for the mapfile. It replaces the default name.

Map Exports

The /MAPINFO option tells the linker to include the specified information in a mapfile, which is created if you specify the `/MAP` option. `EXPORTS` tells the linker to include exported functions.

Debuggable Assembly

/ASSEMBLYDEBUG emits the `DebuggableAttribute` attribute with debug information tracking and disables JIT optimizations.

System Property Page

SubSystem

The /SUBSYSTEM option tells the operating system how to run the `.exe` file. The choice of subsystem affects the entry point symbol (or entry point function) that the linker will choose.

Choices

- **Not Set** - No subsystem set.
- **Console** - Win32 character-mode application. Console applications are given a console by the operating system. If `main` or `wmain` is defined, `CONSOLE` is the default.
- **Windows** - Application doesn't require a console, probably because it creates its own windows for interaction with the user. If `WinMain` or `wWinMain` is defined, `WINDOWS` is the default.
- **Native** - Device drivers for Windows NT. If `/DRIVER:WDM` is specified, `NATIVE` is the default.
- **EFI Application** - EFI Application.
- **EFI Boot Service Driver** - EFI Boot Service Driver.
- **EFI ROM** - EFI ROM.
- **EFI Runtime** - EFI Runtime.

- **POSIX** - Application that runs with the POSIX subsystem in Windows NT.

Minimum Required Version

Specify the minimum required version of the subsystem. The arguments are decimal numbers in the range 0 through 65535.

Heap Reserve Size

Specifies total heap allocation size in virtual memory. Default is 1 MB. (/HEAP:reserve)

Heap Commit Size

Specifies total heap allocation size in physical memory. Default is 4 KB. (`[/HEAP:reserve,commit]`(heap-set-heap-size.md))

Stack Reserve Size

Specifies the total stack allocation size in virtual memory. Default is 1 MB. (/STACK:reserve)

Stack Commit Size

Specifies the total stack allocation size in physical memory. Default is 4 KB. (/STACK:reserve,commit)

Enable Large Addresses

The /LARGEADDRESSAWARE option tells the linker that the application can handle addresses larger than 2 gigabytes. By default, **`/LARGEADDRESSAWARE:NO`** is enabled if **`/LARGEADDRESSAWARE`** isn't otherwise specified on the linker line.

Terminal Server

The /TSAWARE option sets a flag in the `IMAGE_OPTIONAL_HEADER` `DllCharacteristics` field in the program image's optional header. When this flag is set, Terminal Server won't make certain changes to the application.

Swap Run From CD

The /SWAPRUN option tells the operating system to first copy the linker output to a swap file, and then run the image from there. This option is a Windows NT 4.0 (and later) feature. When `CD` is specified, the operating system will copy the image on a removable disk to a page file, and then load it.

Swap Run From Network

The /SWAPRUN option tells the operating system to first copy the linker output to a swap file, and then run the image from there. This option is a Windows NT 4.0 (and later) feature. If `NET` is specified, the operating system will first copy the binary image from the network to a swap file and load it from there. This option is useful for running applications over the network.

Driver

Use the /DRIVER linker option to build a Windows NT kernel mode driver.

Choices

- **Not Set** - Default driver setting.
- **Driver** - Driver
- **UP Only** - `/DRIVER:UPONLY` causes the linker to add the `IMAGE_FILE_UP_SYSTEM_ONLY` bit to the characteristics in the output header to specify that it's a uniprocessor (UP) driver. The operating system will refuse to load a UP driver on a multiprocessor (MP) system.
- **WDM** - `/DRIVER:WDM` causes the linker to set the `IMAGE_DLLCHARACTERISTICS_WDM_DRIVER` bit in the optional header's `DllCharacteristics` field.

Optimization Property Page

References

/OPT:REF eliminates functions and/or data that's never referenced while `/OPT:NOREF` keeps functions and/or data that's never referenced.

Enable COMDAT Folding

Use /OPT:ICF[=iterations] to perform identical COMDAT folding.

Function Order

The /ORDER option tells LINK to optimize your program by placing certain COMDATs into the image in a predetermined order. LINK places the functions in the specified order within each section in the image.

Profile Guided Database

Specify the `.pgd` file for profile guided optimizations. (/PGD)

Link Time Code Generation

Specifies link-time code generation. (/LTCG)

Choices

- **Default** - Default LTCG setting.
- **Use Fast Link Time Code Generation** - Use Link Time Code Generation with /FASTGENPROFILE.
- **Use Link Time Code Generation** - Use Link Time Code Generation.
- **Profile Guided Optimization - Instrument** - Use profile guided optimization with `:PGINSTRUMENT`.
- **Profile Guided Optimization - Optimization** - Specifies that the linker should use the profile data created after running the instrumented binary to create an optimized image.
- **Profile Guided Optimization - Update** - Allows and tracks list of input files to be added or modified from what was specified in the `:PGINSTRUMENT` phase.

Embedded IDL Property Page

MIDL Commands

Specify MIDL command line options. (/MIDL:@responsefile)

Ignore Embedded IDL

The /IGNOREIDL option specifies that any IDL attributes in source code shouldn't be processed into an `.idl` file.

Merged IDL Base File Name

The /IDLOUT option specifies the name and extension of the `.idl` file.

Type Library

The /TLBOUT option specifies the name and extension of the `.tlb` file.

TypeLib Resource ID

Allows you to specify the resource ID of the linker-generated type library. (/TLBID:id)

Windows Metadata Property Page

Generate Windows Metadata

Enables or disables generation of Windows Metadata.

Choices

- **Yes** - Enable generation of Windows Metadata files.
- **No** - Disable the generation of Windows Metadata files.

Windows Metadata File

The /WINMDFILE option switch.

Windows Metadata Key File

Specify a key or key pair to sign the Windows Metadata. (/WINMDKEYFILE:filename)

Windows Metadata Key Container

Specify a key container to sign the Windows Metadata. (/WINMDKEYCONTAINER:name)

Windows Metadata Delay Sign

Partially sign the Windows Metadata. Use /WINMDDELAYSIGN if you only want to place the public key in the Windows Metadata. The default is `/WINMDDELAYSIGN:NO`.

Advanced Property Page

Entry Point

The `[/ENTRY](entry-entry-point-symbol.md)` option specifies an entry point function as the starting address for an `.exe` file or DLL.

No Entry Point

The /NOENTRY option is required for creating a resource-only DLL. Use this option to prevent LINK from linking a reference to `_main` into the DLL.

Set Checksum

The /RELEASE option sets the Checksum in the header of an `.exe` file.

Base Address

Sets a base address for the program. (/BASE:{address[,size] | @filename,key})

Randomized Base Address

Randomized Base Address. (/DYNAMICBASE[:NO])

Fixed Base Address

Creates a program that can be loaded only at its preferred base address. (/FIXED[:NO])

Data Execution Prevention (DEP)

Marks an executable as having been tested to be compatible with Windows Data Execution Prevention feature. (/NXCOMPAT[:NO])

Turn Off Assembly Generation

The /NOASSEMBLY option tells the linker to create an image for the current output file without a .NET Framework assembly.

Unload delay loaded DLL

The **`UNLOAD`** qualifier tells the delay-load helper function to support explicit unloading of the DLL. (/DELAY:UNLOAD)

Nobind delay loaded DLL

The `NOBIND` qualifier tells the linker not to include a bindable Import Address Table (IAT) in the final image. The default is to create the bindable IAT for delay-loaded DLLs. (/DELAY:NOBIND)

Import Library

Overrides the default import library name. (/IMPLIB:filename)

Merge Sections

The /MERGE option combines the first section with the second section, and gives the resulting section the second section name. For example, `/merge:.rdata=.text` merges the `.rdata` section with the `.text` section, and names the combined section `.text`.

Target Machine

The /MACHINE option specifies the target platform for the program.

Choices

- **Not Set**
- **MachineARM**
- **MachineARM64**
- **MachineEBC**
- **MachineIA64**
- **MachineMIPS**
- **MachineMIPS16**
- **MachineMIPSFPU**
- **MachineMIPSFPU16**
- **MachineSH4**
- **MachineTHUMB**
- **MachineX64**
- **MachineX86**

Profile

Produces an output file that can be used with the Performance Tools profiler. Requires the **Generate Debug Info** property be set to **GenerateDebugInformation (/DEBUG)**. (/PROFILE)

CLR Thread Attribute

Explicitly specify the threading attribute for the entry point of your CLR program.

Choices

- **MTA threading attribute** - Applies the MTAThreadAttribute attribute to the entry point of your program.
- **STA threading attribute** - Applies the STAThreadAttribute attribute to the entry point of your program.
- **Default threading attribute** - Same as not specifying /CLRTHREADATTRIBUTE. Lets the Common Language Runtime (CLR) set the default threading attribute.

CLR Image Type

Sets the type (IJW, pure, or safe) of a CLR image.

Choices

- **Force IJW image**
- **Force Pure IL Image**
- **Force Safe IL Image**
- **Default image type**

Key File

Specify key or key pair to sign an assembly. (/KEYFILE:filename)

Key Container

Specify a key container to sign an assembly. (/KEYCONTAINER:name)

Delay Sign

Partially sign an assembly. Use /DELAYSIGN if you only want to place the public key in the assembly. The default is `/DELAYSIGN:NO`.

CLR Unmanaged Code Check

/CLRUNMANAGEDCODECHECK specifies whether the linker will apply `SuppressUnmanagedCodeSecurityAttribute` to linker-generated P/Invoke calls from managed code into native DLLs.

Error Reporting

Allows you to provide internal compiler error (ICE) information directly to the Visual Studio C++ team.

Choices

- **PromptImmediately** - Prompt immediately.
- **Queue For Next Login** - Queue for next sign-in.
- **Send Error Report** - Send error report.
- **No Error Report** - No error report.

SectionAlignment

The /ALIGN option specifies the alignment of each section within the linear address space of the program. The number argument is in bytes and must be a power of two.

Preserve Last Error Code for PInvoke Calls

/CLRSUPPORTLASTERROR, which is on by default, preserves the last error code of functions called through the P/Invoke mechanism, which allows you to call native functions in DLLS, from code compiled with `/clr`.

Choices

- **Enabled** - Enable `/CLRSupportLastError`.
- **Disabled** - Disable `/CLRSupportLastError`.
- **System DLLs Only** - Enable `/CLRSupportLastError` for system DLLs only.

Image Has Safe Exception Handlers

When /SAFESEH is specified, the linker will only produce an image if it can also produce a table of the image's safe exception handlers. This table specifies for the operating system which exception handlers are valid for the image.

Command line property pages

Article • 09/22/2022

Most property page folders that correspond with a command-line tool contain a **Command Line** property page. For information on how to access the **Command Line** property pages, see Set compiler and build properties.

Command Line property page

All Options

The **All Options** display-only control shows the tool command line created by the properties set in the folder.

Additional Options

This property's edit control lets you specify other command-line options that are valid for the tool but that don't have a corresponding property.

The options that you enter in the edit box are passed through to the tool for the folder after the options listed in **All Options**. No verification or validity checks are done on the options you enter, and there's no dependency checking.

See also

Windows C++ project property page reference
Linux C++ property page reference
Linker property pages
Manifest tool property pages
MIDL property pages
NMake property page
XML document generator tool property pages

NMake Property Page

Article • 02/07/2023

The **NMake** property page lets you specify build settings for *Makefile* projects. (NMAKE is the Microsoft implementation of Make.)

For more information about Makefile projects, see Creating a Makefile Project. For non-Windows Makefile projects, see Makefile Project Properties (Linux C++), General Project Properties (Android C++ Makefile) or NMake Properties (Android C++).

The property page contains the following properties:

General

- **Build Command Line**

 Specifies the command to be run when **Build** is clicked on the **Build** menu.

- **Rebuild All Command Line**

 Specifies the command to be run when **Rebuild All** is clicked on the **Build** menu.

- **Clean Command Line**

 Specifies the command to be run when **Clean** is clicked on the **Build** menu.

- **Output**

 Specifies the name of the file that will contain the output for the command line. By default, this file name is based on the project name.

IntelliSense

- **Preprocessor Definitions**

 Specifies any preprocessor definitions that the source files use. The default value is determined by the current platform and configuration.

- **Include Search Path**

 Specifies the directories where the compiler searches for include files.

- **Forced Includes**

Specifies files that the preprocessor automatically processes even if they aren't included in the project files.

- **Assembly Search Path**

 Specifies the directories where the .NET Framework searches when it resolves .NET assemblies.

- **Forced Using Assemblies**

 Specifies assemblies that the .NET Framework automatically processes.

- **Additional Options**

 Specifies any extra compiler switches for IntelliSense to use when it parses C++ files.

For information about how to access this property page, see Set C++ compiler and build properties in Visual Studio.

For information about how to programmatically access members of this object, see VCNMakeTool.

See also

C++ project property page reference

Manifest Tool Property Pages

Article • 09/22/2022

Use these pages to specify general options for Mt.exe. These pages are found under **Project** > **Properties** > **Configuration Properties** > **Manifest Tool**.

General Property Page

Suppress Startup Banner

Yes (`/nologo`) specifies that standard Microsoft copyright data will be concealed when the manifest tool is started. Use this option to suppress unwanted output in log files when you run `mt.exe`, either as part of a build process or from a build environment.

Verbose Output

Yes (`/verbose`) specifies that more build information will be displayed during manifest generation.

Assembly Identity

Uses the `/identity` option to specify an identity string, which holds the attributes for the <assemblyIdentity> element. An identity string begins with the value for the `name` attribute, and is followed by *attribute* = *value* pairs. The attributes in an identity string are delimited by a comma.

Here's an example identity string:

```
Microsoft.Windows.Common-Controls, processorArchitecture=x86, version=6.0.0.0, type=win32, publicKeyToken=6595b64144ccf1df
```

Input and Output Property Page

Additional Manifest Files

Uses the `/manifest` option to specify the full paths of more manifest files that the manifest tool will process or merge. Full paths are delimited by a semicolon. (`/manifest [manifest1] [manifest2] ...`)

Input Resource Manifests

Uses the **`/inputresource`** option to specify the full path of a resource of type `RT_MANIFEST`, to input into the manifest tool. The path can be followed by the specified resource ID. For example:

`dll_with_manifest.dll;#1`

Embed Manifest

- **Yes** specifies that the project system will embed the application manifest file into the assembly.
- **No** specifies that the project system will create the application manifest file as a stand-alone file.

Output Manifest File

Specifies the name of the output manifest file. This property is optional when only one manifest file is operated upon by the manifest tool. (`/out:[file];#[resource ID]`)

Manifest Resource File

Specifies the output resources file used to embed the manifest into the project output.

Generate Catalog Files

Uses the **`/makecdfs`** option to specify that the manifest tool will generate catalog definition files (*`.cdf`* files), which are used to make catalogs. (`/makecdfs`)

Generate Manifest From ManagedAssembly

Generates a manifest from a managed assembly. (`/managedassemblyname:[file]`)

Suppress Dependency Element

Used with **`/managedassemblyname`**. Suppresses the generation of dependency elements in the final manifest. (`/nodependency`)

Generate Category Tags

Used with `/managedassemblyname`. `/category` causes the category tags to be generated. (`/category`)

DPI Awareness

Specifies whether the application is DPI-aware. By default, the setting is **Yes** for MFC projects and **No** otherwise because only MFC projects have built in DPI awareness. You can override the setting to **Yes** if you add code to handle different DPI settings. Your application might appear fuzzy or small if it isn't DPI-aware, but you set a DPI-aware option.

Choices

- **None**
- **High DPI Aware**
- **Per Monitor High DPI Aware**

Isolated COM Property Page

For more information about isolated COM, see Isolated applications and How to: Build isolated applications to consume COM components.

Type Library File

Specifies the type library to use for regfree COM manifest support. (`/tlb:[file]`)

Registrar Script File

Specifies the registrar script file to use for regfree COM manifest support. (`/rgs:[file]`)

Component File Name

Specifies the file name of the component that is built from the .tlb or .rgs specified. (`/dll:[file]`)

Replacements File

Specifies the file that contains values for replaceable strings in the RGS file. (`/replacements:[file]`)

Advanced Property Page

Update File Hashes

Computes the hash of files specified in the `file` elements, and then updates the hash attribute with this value. (`/hashupdate:[path]`)

Update File Hashes Search Path

Specifies the search path to use when updating the file hashes.

Additional Options

Allows you to specify more options.

See also

C++ project property page reference

Resources property page

Article • 09/22/2022

For native Windows desktop programs, the build invokes the Resource Compiler (rc.exe) to add images, string tables, and *.res* files to the binary. The properties exposed in this property page are passed to the Resource Compiler, not to the C++ compiler or the linker. For more information on the properties listed here and how they map to RC command-line options, see Using RC (The RC Command Line). For information on how to access the **Resources** property pages, see Set C++ compiler and build properties in Visual Studio. To programmatically access these properties, see VCResourceCompilerTool.

Properties for .NET resources in C++/CLI applications are exposed in the Managed Resources Property Page.

Preprocessor Definitions

Specifies one or more defines for the resource compiler. (/d[macro])

Undefine Preprocessor Definitions

Undefine a symbol. (/u)

Culture

Lists the culture (such as US English or Italian) used in the resources. (/l [num])

Additional Include Directories

Specifies one or more directories to add to the include path; use semi-colon delimiter if more than one. (/I[path])

Ignore Standard Include Paths

Prevents the resource compiler from searching for include files in directories specified in the INCLUDE environment variables. (/X)

Show Progress

Send progress messages to output window. (/v)

Suppress Startup Banner

Suppress the display of the startup banner and information message (/nologo)

Resource File Name

Specifies the name of the resource file (/fo[file])

Null Terminate Strings

Append null's to all strings in the string tables. (/n)

See also

C++ project property page reference

Managed Resources Property Page

Article • 08/03/2021

The **Managed Resources** property page exposes the following properties for the managed resource compiler resgen.exe when using .NET resources in C++/CLI programs:

- **Resource Logical Name**

 Specifies the *logical name* of the generated intermediate .resources file. The logical name is the name used to load the resource. If no logical name is specified, the resource (.resx) file name is used as the logical name.

- **Output File Name**

 Specifies the name of the final output file that the resource (.resx) file contributes to.

- **Default Localized Resources**

 Specifies whether the given .resx file contributes to the default resources or to a satellite .dll.

For information on how to access the **Managed Resources** property page, see Set C++ compiler and build properties in Visual Studio.

See also

Using RC (The RC Command Line)
C++ project property page reference
/ASSEMBLYRESOURCE (Embed a Managed Resource)

MIDL Property Pages

Article • 09/22/2022

The MIDL property pages are available as an item property on an .IDL file in a C++ project that uses COM. Use them to configure the MIDL Compiler. For information on how to programmatically access MIDL options for C++ projects, see VCMidlTool object. See also General MIDL Command-line Syntax.

General Property Page

Preprocessor Definitions

Specifies one or more defines, including MIDL macros (/D)[macros]).

Additional Include Directories

Specifies one or more directories to add to the include path (/I[path]).

Additional Metadata Directories

Specify the directory containing the Windows.Foundation.WinMD file (/metadata_dir [path]).

Enable Windows Runtime

Enable Windows Runtime semantics to create Windows metadata file (/winrt).

Ignore Standard Include Path

Ignore the current and the INCLUDE directories (/no_def_idir).

MkTypLib Compatible

Forces compatibility with mktyplib.exe version 2.03 (/mktyplib203).

Warning Level

Selects the strictness of the MIDL code errors (/W).

Choices

- 1
- 1
- 2
- 3
- 4

Treat Warnings as Errors

Enables MIDL to treat all warnings as errors (/WX).

Suppress Startup Banner

Suppress the display of the startup banner and information message (/nologo).

C Compiler Char Type

Specifies the default character type of the C compiler that will be used to compile the generated code. (/char signed|unsigned|ascii7).

Choices

- **Signed** - Signed
- **Unsigned** - Unsigned
- **Ascii** - Ascii

Target Environment

Specifies which environment to target (/env arm32|win32|ia64|x64).

Choices

- **Not Set** - Win32
- **Microsoft Windows 32-bit** - Win32
- **Microsoft Windows 64-bit on Itanium** - IA64
- **Microsoft Windows ARM** - ARM
- **Microsoft Windows ARM64** - ARM64
- **Microsoft Windows 64-bit on x64** - X64

Generate Stubless Proxies

Generate fully interpreted stubs with extensions and stubless proxies for object interfaces (/Oicf, /Oif).

Suppress Compiler Warnings

Suppress compiler warning messages (/no_warn).

Application Configuration Mode

Allow selected ACF attributes in the IDL file (/app_config).

Locale ID

Specifies the LCID for input files, file names and directory paths (/lcid DECIMAL).

Multi-Processor Compilation

Run multiple instances at the same time.

Output Property Page

Output Directory

Specifies the output directory (/out [directory]).

Metadata File

Specifies the name of the generated metadata file (/winmd filename).

Header File

Specifies the name of the generated header file (/h filename).

DllData File

Specifies the name of the DLLDATA file (/dlldata filename).

IID File

Specifies the name for the Interface Identifier file (/iid filename).

Proxy File

Specifies the name of the proxy file (/proxy filename).

Generate Type Library

Specify not to generate a type library ([/notlb] for no).

Type Library

Specifies the name of the type library file (/tlb filename).

Generate Client Stub Files

Generate client stub file only (/client [stub|none]).

Choices

- **Stub** - Stub
- **None** - None

Generate Server Stub Files

Generate server stub file only (/server [stub|none]).

Choices

- **Stub** - Stub
- **None** - None

Client Stub File

Specify the client stub file (/cstub [file]).

Server Stub File

Specify the server stub file (/sstub [file]).

Type Library Format

Specifies the type library file format ([/oldtlb|/newtlb]).

Choices

- **NewFormat** - New Format
- **OldFormat** - Old Format

Advanced Property Page

C Preprocess Options

Specifies switches to pass to C compiler preprocessor (/cpp_opt switches).

Undefine Preprocessor Definitions

Specifies one or more undefines, including MIDL macros (/U [macros]).

Enable Error Checking

Select error checking option ([/error all|none]).

Choices

- **EnableCustom** - All
- **All** - All
- **None** - None

Check Allocations

Check for out of memory errors (/error allocation).

Check Bounds

Check size vs transmission length specification (/error bounds_check).

Check Enum Range

Check enum values to be in allowable range (/error enum).

Check Reference Pointers

Check ref pointers to be non-null (/error ref).

Check Stub Data

Emit additional check for server side stub data validity (/error stub_data).

Prepend with 'ABI' namespace

Prepend the 'ABI' namespace to all types. (/ns_prefix).

Validate Parameters

Generate additional information to validate parameters (/robust | /no_robust).

Struct Member Alignment

Specifies the packing level of structures in the target system (/ZpN).

Choices

- **Not Set** - Not Set
- **1 Byte** - Zp1
- **2 Byte** - Zp2
- **4 Byte** - Zp4
- **8 Byte** - Zp8

Redirect Output

Redirects output from screen to a file (/o file).

Minimum Target System

Set the minimum target system (/target STRING).

Web References Property Page

Article • 08/03/2021

The **Web References** property page specifies how the XML Web service proxy class will be generated. An XML Web service proxy class will be generated if you add a web reference to your project.

The **Web References** property page contains the following properties:

- **Output file**

 The name of the file to contain the XML Web service proxy class.

- **Suppress Startup Banner**

 Do not display the banner for the Web Services Description Language Tool (Wsdl.exe).

- **Namespace**

 Specifies the name of the generated web proxy.

- **Additional References**

 Specifies the additional DLLs referenced by the proxy DLL.

For information on how to access the **Web Reference** property page, see Set C++ compiler and build properties in Visual Studio.

See also

C++ project property page reference

XML Data Generator Tool Property Page

Article • 08/03/2021

The **XML Data Generator Tool** property page becomes available when you add a dataset to a project.

The **XML Data Generator Tool** property page contains the following properties:

- **Output File**

 Specifies the output file name to use.

- **Suppress Startup Banner**

 Suppresses the display of the startup banner and information messages.

- **Generated Proxy Language**

 Determines whether or not to emit managed code.

For information on how to access the **XML Data Generator Tool** property page, see Set C++ compiler and build properties in Visual Studio.

For information on how to programmatically access members of this object, see VCXMLDataGeneratorTool

See also

C++ project property page reference

XML Document Generator Tool Property Pages

Article • 03/03/2022

The XML Document Generator Tool property page exposes the functionality of `xdcmake.exe`, or XDCMake. XDCMake merges `.xdc` files into an `.xml` file when your source code contains documentation comments and /doc (Process Documentation Comments) (C/C++) is specified. For more information on adding documentation comments to source code, see Recommended tags for documentation comments.

> **Note**
>
> XDCMake options in the development environment (property pages) differ from the options when `xdcmake.exe` is used at the command line. For information on using `xdcmake.exe` at the command line, see **XDCMake reference**.

UIElement List

- **Suppress Startup Banner**

 Suppress copyright message.

- **Additional Document Files**

 Additional directories in which you want the project system to look for `.xdc` files. XDCMake always looks for `.xdc` files generated by the project. Multiple directories can be specified.

- **Output Document File**

 The name and directory location of the `.xml` output file. For more information on using macros to specify directory locations, see Common macros for build commands and properties.

- **Document Library Dependencies**

 If your project has a dependency on a `.lib` project in the solution, you can process `.xdc` files from the `.lib` project into the `.xml` files for the current project.

See also

C++ project property page reference

Custom Build Step Property Page: General

Article • 08/03/2021

For each project configuration and target platform combination in your project, you can specify a custom step to execute when the project is built.

For the Linux version of this page, see Custom Build Step Properties (Linux C++).

General page

- **Command Line**

 The command to be executed by the custom build step.

- **Description**

 A message that's displayed when the custom build step runs.

- **Outputs**

 The output file that the custom build step generates. This setting is required so that incremental builds work correctly.

- **Additional Dependencies**

 A semicolon-delimited list of any additional input files to use for the custom build step.

- **Execute After and Execute Before**

 These options define when the custom build step is run in the build process, relative to the listed targets. The most commonly listed targets are `BuildGenerateSources`, `BuildCompile`, and `BuildLink`, because they represent the major steps in the build process. Other often-listed targets are `Midl`, `CLCompile`, and `Link`.

- **Treat Output As Content**

 This option is only meaningful for Universal Windows Platform or Windows Phone apps, which include all content files in the `.appx` package.

To specify a custom build step

1. On the menu bar, choose **Project** > **Properties** to open the **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Custom Build Step** > **General** page.

3. Modify the settings.

See also

C++ project property page reference

HLSL Compiler Property Pages

Article • 08/03/2021

You can use the HLSL compiler (fxc.exe) property pages to configure how individual HLSL shader files are built. You can also specify command-line arguments to the HLSL compiler by using the **Additional Options** property of the **Command Line** property page; this includes arguments that can't be configured by using other properties of the HLSL property pages. For information about the HLSL compiler, see Effect-Compiler Tool

HLSL General Property Page

Additional Include Directories

Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

Entrypoint Name

Specifies the name of the entry point for the shader (/E[name])

Disable Optimizations

Yes (/Od) to disable optimizations; otherwise, **No**. By default, the value is **Yes (/Od)** for **Debug** configurations and **No** for **Release** configurations.

The **/Od** command-line argument to the HLSL compiler implicitly applies the **/Gfp** command-line argument, but output may not be identical to output that is produced by passing both the **/Od** and **/Gfp** command-line arguments explicitly.

Enable Debugging Information

Yes (/Zi) to enable debugging information; otherwise, **No**. By default, the value is **Yes (/Zi)** for **Debug** configurations and **No** for **Release** configurations.

Shader Type

Specifies the kind of shader. Different kinds of shaders implement different parts of the graphics pipeline. Certain kinds of shaders are available only in more recent shader

models (which are specified by the **Shader Model** property)—for example, compute shaders were introduced in shader model 5.

This property corresponds to the **[type]** portion of the **/T [type]_[model]** command-line argument to the HLSL compiler. The **Shader Models** property specifies the **[model]** portion of the argument.

Choices

- **Effect**
- **Vertex Shader**
- **Pixel Shader**
- **Geometry Shader**
- **Hull Shader**
- **Domain Shader**
- **Compute Shader**
- **Library**
- **Generate Root Signature Object**

Shader Model

Specifies the shader model. Different shader models have different capabilities. In general, more recent shader models offer expanded capabilities but require more modern graphics hardware to run the shader code. Certain kinds of shaders (which are specified by the **Shader Type** property) are available only in more recent shader models —for example, compute shaders were introduced in shader model 5.

This property corresponds to the **[model]** portion of the **/T [type]_[model]** command-line argument to the HLSL compiler. The **Shader Type** property specifies the **[type]** portion of the argument.

All Resources Bound

Compiler will assume that all resources that a shader may reference are bound and are in good state for the duration of shader execution (/all_resources_bound). Available for Shader Model 5.1 and above.

Enable Unbounded Descriptor Tables

Inform the compiler that a shader may contain a declaration of a resource array with unbounded range (/enable_unbounded_descriptor_tables). Available for Shader Model 5.1 and above.

Set Root Signature

Attach root signature to shader bytecode (/setrootsignature). Available for Shader Model 5.0 and above.

Preprocessor Definitions

Adds one or more preprocessor symbol definitions to apply to the HLSL source code file. Use semi-colons to separate the symbol definitions.

This property corresponds to the **/D [definitions]** command-line argument to the HLSL compiler.

Compile a Direct2D custom pixel shader effect

Compile a Direct2D custom effect that contains pixel shaders. Do not use for a vertex or compute custom effect.

Multi Processor Compilation

Run multiple instances at the same time.

Advanced Property Page

Suppress Startup Banner

Suppresses the display of the startup banner and information message. (/nologo)

Treat Warnings As Errors

Treats all compiler warnings as errors. For a new project, it may be best to use /WX in all compilations; resolving all warnings will ensure the fewest possible hard-to-find code defects.

Output Files Property Page

Header Variable Name

Specifies a name for the variable name in the header file (/Vn [name])

Header File Name

Specifies a name for header file containing object code. (/Fh [name])

Object File Name

Specifies a name for object file. (/Fo [name])

Assembler Output

Specifies the contents of assembly language output file. (/Fc, /Fx)

Choices

- **No Listing** - No listing.
- **Assembly-Only Listing** - Assembly code file
- **Assembly Code and Hex** - Assembly code and hex listing file

Assembler Output File

Specifies file name for assembly code listing file

See also

C++ project property page reference
Command Line Property Pages
Compiling Shaders

Compiling a C/C++ project

Article • 03/03/2022

C and C++ compiler options can be set either in the Visual Studio IDE or on the command line.

In Visual Studio

You can set compiler options for each project in its Visual Studio **Property Pages** dialog box. In the left pane, select **Configuration Properties**, **C/C++** and then choose the compiler option category. The topic for each compiler option describes how it can be set and where it is found in the development environment. For more information and a complete list of options, see MSVC compiler options.

From the command line

You can set compiler (CL.exe) options:

- On the command line
- In command files
- In the CL environment variable

Options specified in the CL environment variable are used every time you invoke CL. If a command file is named in the CL environment variable or on the command line, the options specified in the command file are used. Unlike either the command line or the CL environment variable, a command file allows you to use multiple lines of options and filenames.

Compiler options are processed "left to right," and when a conflict is detected, the last (rightmost) option wins. The CL environment variable is processed before the command line, so in any conflicts between CL and the command line, the command line takes precedence.

Additional Compiler Topics

- MSVC Compiler Options
- Precompiled Header Files

- CL Invokes the Linker

For information on choosing the compiler host and target architecture, see Configure C++ projects for 64-bit, x64 targets.

See also

C/C++ Building Reference

Compiler Command-Line Syntax

Article • 03/03/2022

The CL command line uses the following syntax:

```
CL [option...] file... [option | file]... [lib...] [@command-file] [/link
link-opt...]
```

The following table describes input to the CL command.

Entry	Meaning
option	One or more CL options. Note that all options apply to all specified source files. Options are specified by either a forward slash (/) or a dash (-). If an option takes an argument, the option's description documents whether a space is allowed between the option and the arguments. Option names (except for the /HELP option) are case sensitive. For more information, see Order of CL Options.
`file`	The name of one or more source files, .obj files, or libraries. CL compiles source files and passes the names of the .obj files and libraries to the linker. For more information, see CL Filename Syntax.
lib	One or more library names. CL passes these names to the linker.
command-file	A file that contains multiple options and filenames. For more information, see CL Command Files.
link-opt	One or more MSVC Linker Options. CL passes these options to the linker.

You can specify any number of options, filenames, and library names, as long as the number of characters on the command line does not exceed 1024, the limit dictated by the operating system.

For information about the return value of cl.exe, see Return Value of cl.exe .

> **Note**
>
> The command-line input limit of 1024 characters is not guaranteed to remain the same in future releases of Windows.

See also

CL Filename Syntax

Article • 08/03/2021

CL accepts files with names that follow FAT, HPFS, or NTFS naming conventions. Any filename can include a full or partial path. A full path includes a drive name and one or more directory names. CL accepts filenames separated either by backslashes (\) or forward slashes (/). Filenames that contain spaces must be surrounded by double quote characters. A partial path omits the drive name, which CL assumes to be the current drive. If you don't specify a path, CL assumes the file is in the current directory.

The filename extension determines how files are processed. C and C++ files, which have the extension .c, .cxx, or .cpp, are compiled. Other files, including .obj files, libraries (.lib), and module-definition (.def) files, are passed to the linker without being processed.

See also

MSVC Compiler Command-Line Syntax

Order of CL Options

Article • 08/03/2021

Options can appear anywhere on the CL command line, except for the /link option, which must occur last. The compiler begins with options specified in the CL environment variable and then reads the command line from left to right — processing command files in the order it encounters them. Each option applies to all files on the command line. If CL encounters conflicting options, it uses the rightmost option.

See also

MSVC Compiler Command-Line Syntax

Return Value of cl.exe

Article • 11/11/2021

cl.exe returns zero for success (no errors) and non-zero otherwise.

The return value of cl.exe can be useful if you are compiling from a script, powershell, .cmd, or .bat file. We recommend that you capture the output of the compiler in case there are errors or warnings, so that you can resolve them.

There are too many possible error exit codes for cl.exe to list them all. You can look up an error code in the winerror.h or ntstatus.h files included in the Windows Software Development Kit in the %ProgramFiles(x86)%\Windows Kits*version*\Include\shared\ directory. Error codes returned in decimal must be converted to hexadecimal for search. For example, an error code of -1073741620 converted to hexadecimal is 0xC00000CC. This error is found in ntstatus.h, where the corresponding message is "The specified share name cannot be found on the remote server." For a downloadable list of Windows error codes, see [MS-ERREF] Windows Error Codes.

You can also use the error lookup utility in Visual Studio to find out what a compiler error message means. In a Visual Studio command shell, enter **errlook.exe** to start the utility; or in the Visual Studio IDE, on the menu bar, choose **Tools**, **Error Lookup**. Enter the error value to find the descriptive text associated with the error. For more information see ERRLOOK Reference.

Remarks

The following is a sample .bat file that uses the return value of cl.exe.

Windows Command Prompt

```
echo off
cl /W4 t.cpp
@if ERRORLEVEL == 0 (
   goto good
)

@if ERRORLEVEL != 0 (
   goto bad
)

:good
   echo "clean compile"
   echo %ERRORLEVEL%
   goto end
```

```
:bad
   echo "error or warning"
   echo %ERRORLEVEL%
   goto end

:end
```

See also

MSVC Compiler Command-Line Syntax

CL environment variables

Article • 08/03/2021

The CL tool uses the following environment variables:

- CL and _CL_, if defined. The CL tool prepends the options and arguments defined in the CL environment variable to the command-line arguments, and appends the options and arguments defined in _CL_, before processing.
- INCLUDE, which must point to the \include subdirectory of your Visual Studio installation.
- LIBPATH, which specifies directories to search for metadata files referenced with #using. For more information on LIBPATH, see #using.

You can set the CL or _CL_ environment variable using the following syntax:

> SET CL=[[*option*] ... [*file*] ...] [/link *link-opt* ...]
> SET _CL_=[[*option*] ... [*file*] ...] [/link *link-opt* ...]

For details on the arguments to the CL and _CL_ environment variables, see MSVC Compiler Command-Line Syntax.

You can use these environment variables to define the files and options you use most often. Then use the command line to give more files and options to CL for specific purposes. The CL and _CL_ environment variables are limited to 1024 characters (the command-line input limit).

You can't use the /D option to define a symbol that uses an equal sign (=). Instead, you can use the number sign (#) for an equal sign. In this way, you can use the CL or _CL_ environment variables to define preprocessor constants with explicit values—for example, `/DDEBUG#1` to define `DEBUG=1`.

For more information, see Use the MSVC toolset from the command line.

Examples

The following command is an example of setting the CL environment variable:

> SET CL=/Zp2 /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ

When the CL environment variable is set, if you enter `CL INPUT.C` at the command line, the effective command becomes:

```
CL /Zp2 /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ INPUT.C
```

The following example causes a plain CL command to compile the source files FILE1.c and FILE2.c, and then link the object files FILE1.obj, FILE2.obj, and FILE3.obj:

```
SET CL=FILE1.C FILE2.C
SET _CL_=FILE3.OBJ
CL
```

These environment variables make the call to CL have the same effect as the following command line:

```
CL FILE1.C FILE2.C FILE3.OBJ
```

See also

Setting Compiler Options
MSVC Compiler Options

CL Command Files

Article • 08/03/2021

A command file is a text file that contains compiler options and filenames. It supplies options you would otherwise type on the command line, or specify using the CL environment variable. CL accepts a compiler command file as an argument, either in the CL environment variable, or on the command line. Unlike either the command line or the CL environment variable, you can use multiple lines of options and filenames in a command file.

Options and filenames in a command file are processed when a command filename appears within the CL environment variable or on the command line. However, if the `/link` option appears in the command file, all options on the rest of the line are passed to the linker. Options in later lines in the command file, and options on the command line after the command file invocation, are still accepted as compiler options. For more information on how the order of options affects their interpretation, see Order of CL Options.

A command file must not contain the CL command. Each option must begin and end on the same line; you can't use the backslash (`\`) to combine an option across two lines.

A command file is specified by an at sign (`@`) followed by a filename. The filename can specify an absolute or relative path.

For example, if the following command is in a file named RESP:

Windows Command Prompt

```
/Ot /link LIBC.LIB
```

and you specify the following CL command:

Windows Command Prompt

```
CL /Ob2 @RESP MYAPP.C
```

the command to CL is as follows:

Windows Command Prompt

```
CL /Ob2 /Ot MYAPP.C /link LIBC.LIB
```

Here you can see how the command line and the command-file commands are effectively combined.

See also

MSVC compiler command-line syntax
MSVC compiler options

CL Invokes the Linker

Article • 08/03/2021

CL automatically invokes the linker after compiling unless the /c option is used. CL passes to the linker the names of .obj files created during compiling and the names of any other files specified on the command line. The linker uses the options listed in the LINK environment variable. You can use the /link option to specify linker options on the CL command line. Options that follow the /link option override those in the LINK environment variable. The options in the following table suppress linking.

Option	Description
/c	Compile without linking
/E, /EP, /P	Preprocess without compiling or linking
/Zg	Generate function prototypes
/Zs	Check syntax

For further details about linking, see MSVC Linker Options.

Example

Assume that you are compiling three C source files: MAIN.c, MOD1.c, and MOD2.c. Each file includes a call to a function defined in a different file:

- MAIN.c calls the function `func1` in MOD1.c and the function `func2` in MOD2.c.
- MOD1.c calls the standard library functions `printf_s` and `scanf_s`.
- MOD2.c calls graphics functions named `myline` and `mycircle`, which are defined in a library named MYGRAPH.lib.

To build this program, compile with the following command line:

```
CL MAIN.c MOD1.C MOD2.C MYGRAPH.lib
```

CL first compiles the C source files and creates the object files MAIN.obj, MOD1.obj, and MOD2.obj. The compiler places the name of the standard library in each .obj file. For more details, see Use Run-Time Library.

CL passes the names of the .obj files, along with the name MYGRAPH.lib, to the linker. The linker resolves the external references as follows:

1. In MAIN.obj, the reference to `func1` is resolved using the definition in MOD1.obj; the reference to `func2` is resolved using the definition in MOD2.obj.

2. In MOD1.obj, the references to `printf_s` and `scanf_s` are resolved using the definitions in the library that the linker finds named within MOD1.obj.

3. In MOD2.obj, the references to `myline` and `mycircle` are resolved using the definitions in MYGRAPH.lib.

See also

MSVC Compiler Options
Setting Compiler Options

Compiler Options

Article • 08/03/2021

cl.exe is a tool that controls the Microsoft C++ (MSVC) C and C++ compilers and linker. cl.exe can be run only on operating systems that support Microsoft Visual Studio for Windows.

> **Note**
>
> You can start this tool only from a Visual Studio developer command prompt. You cannot start it from a system command prompt or from File Explorer. For more information, see **Use the MSVC toolset from the command line**.

The compilers produce Common Object File Format (COFF) object (.obj) files. The linker produces executable (.exe) files or dynamic-link libraries (DLLs).

All compiler options are case-sensitive. You may use either a forward slash (`/`) or a dash (`-`) to specify a compiler option.

To compile without linking, use the /c option.

Find a compiler option

To find a particular compiler option, see one of the following lists:

- Compiler Options Listed Alphabetically
- Compiler Options Listed by Category

Specify compiler options

The topic for each compiler option discusses how it can be set in the development environment. For information on specifying options outside the development environment, see:

- MSVC Compiler Command-Line Syntax
- CL Command Files
- CL Environment Variables

Related build tools

MSVC Linker Options also affect how your program is built.

See also

C/C++ Building Reference
CL Invokes the Linker

Compiler options listed by category

Article • 11/13/2023

This article contains a categorical list of compiler options. For an alphabetical list, see Compiler options listed alphabetically.

Optimization

Option	Purpose
/favor:<blend\|AMD64\|INTEL64\|ATOM>	Produces code that is optimized for a specified architecture, or for a range of architectures.
/O1	Creates small code.
/O2	Creates fast code.
/Ob<n>	Controls inline expansion.
/Od	Disables optimization.
/Og	Deprecated. Uses global optimizations.
/Oi[-]	Generates intrinsic functions.
/Os	Favors small code.
/Ot	Favors fast code.
/Ox	A subset of /O2 that doesn't include /GF or /Gy.
/Oy	Omits frame pointer. (x86 only)

Code generation

Option	Purpose
/arch:<IA32\|SSE\|SSE2\|AVX\|AVX2\|AVX512>	Minimum CPU architecture requirements. IA32, SSE, and SSE2 are x86 only.
/clr	Produces an output file to run on the common language runtime.
/clr:implicitKeepAlive-	Turn off implicit emission of `System::GC::KeepAlive(this)`.
/clr:initialAppDomain	Enable initial AppDomain behavior of Visual C++ 2002.

Option	Purpose
/clr:netcore	Produce assemblies targeting .NET Core runtime.
/clr:noAssembly	Don't produce an assembly.
/clr:nostdimport	Don't import any required assemblies implicitly.
/clr:nostdlib	Ignore the system .NET framework directory when searching for assemblies.
/clr:pure	Produce an IL-only output file (no native executable code).
/clr:safe	Produce an IL-only verifiable output file.
/EHa	Enable C++ exception handling (with SEH exceptions).
/EHc	`extern "C"` defaults to `nothrow`.
/EHr	Always generate `noexcept` runtime termination checks.
/EHs	Enable C++ exception handling (no SEH exceptions).
/fp:contract	Consider floating-point contractions when generating code.
/fp:except[-]	Consider floating-point exceptions when generating code.
/fp:fast	"fast" floating-point model; results are less predictable.
/fp:precise	"precise" floating-point model; results are predictable.
/fp:strict	"strict" floating-point model (implies `/fp:except`).
/fpcvt:BC	Backward-compatible floating-point to unsigned integer conversions.
/fpcvt:IA	Intel native floating-point to unsigned integer conversion behavior.
/fsanitize	Enables compilation of sanitizer instrumentation such as AddressSanitizer.
/fsanitize-coverage	Enables compilation of code coverage instrumentation for libraries such as LibFuzzer.
/GA	Optimizes for Windows applications.
/Gd	Uses the `__cdecl` calling convention. (x86 only)
/Ge	Deprecated. Activates stack probes.
/GF	Enables string pooling.

Option	Purpose
/Gh	Calls hook function `_penter`.
/GH	Calls hook function `_pexit`.
/GL[-]	Enables whole program optimization.
/Gm[-]	Deprecated. Enables minimal rebuild.
/Gr	Uses the `__fastcall` calling convention. (x86 only)
/GR[-]	Enables run-time type information (RTTI).
/GS[-]	Checks buffer security.
/Gs[n]	Controls stack probes.
/GT	Supports fiber safety for data allocated by using static thread-local storage.
`/Gu[-]`	Ensure distinct functions have distinct addresses.
/guard:cf[-]	Adds control flow guard security checks.
/guard:ehcont[-]	Enables EH continuation metadata.
/Gv	Uses the `__vectorcall` calling convention. (x86 and x64 only)
/Gw[-]	Enables whole-program global data optimization.
/GX[-]	Deprecated. Enables synchronous exception handling. Use /EH instead.
/Gy[-]	Enables function-level linking.
/Gz	Uses the `__stdcall` calling convention. (x86 only)
/GZ	Deprecated. Enables fast checks. (Same as /RTC1)
/homeparams	Forces parameters passed in registers to be written to their locations on the stack upon function entry. This compiler option is only for the x64 compilers (native and cross compile).
/hotpatch	Creates a hotpatchable image.
/jumptablerdata	Put switch case statement jump tables in the `.rdata` section.
/Qfast_transcendentals	Generates fast transcendentals.

Option	Purpose
/QIfist	Deprecated. Suppresses the call of the helper function `_ftol` when a conversion from a floating-point type to an integral type is required. (x86 only)
/Qimprecise_fwaits	Removes `fwait` commands inside `try` blocks.
/QIntel-jcc-erratum	Mitigates the performance impact of the Intel JCC erratum microcode update.
/Qpar	Enables automatic parallelization of loops.
/Qpar-report:n	Enables reporting levels for automatic parallelization.
/Qsafe_fp_loads	Uses integer move instructions for floating-point values and disables certain floating point load optimizations.
/Qspectre[-]	Enable mitigations for CVE 2017-5753, for a class of Spectre attacks.
/Qspectre-load	Generate serializing instructions for every load instruction.
/Qspectre-load-cf	Generate serializing instructions for every control flow instruction that loads memory.
/Qvec-report:n	Enables reporting levels for automatic vectorization.
/RTC1	Enable fast runtime checks (equivalent to `/RTCsu`).
/RTCc	Convert to smaller type checks at run-time.
/RTCs	Enable stack frame runtime checks.
/RTCu	Enables uninitialized local usage checks.
/volatile:iso	Acquire/release semantics not guaranteed on volatile accesses.
/volatile:ms	Acquire/release semantics guaranteed on volatile accesses.

Output files

Option	Purpose
/doc	Processes documentation comments to an XML file.
/FA	Configures an assembly listing file.

Option	Purpose
/Fa	Creates an assembly listing file.
/Fd	Renames program database file.
/Fe	Renames the executable file.
/Fi	Specifies the preprocessed output file name.
/Fm	Creates a mapfile.
/Fo	Creates an object file.
/Fp	Specifies a precompiled header file name.
/FR, /Fr	Name generated `.sbr` browser files. `/Fr` is deprecated.
`/Ft<dir>`	Location of the header files generated for `#import`.

Preprocessor

Option	Purpose
/AI<dir>	Specifies a directory to search to resolve file references passed to the #using directive.
/C	Preserves comments during preprocessing.
/D<name>{=\|#}<text>	Defines constants and macros.
/E	Copies preprocessor output to standard output.
/EP	Copies preprocessor output to standard output.
/FI<file>	Preprocesses the specified include file.
/FU<file>	Forces the use of a file name, as if it had been passed to the #using directive.
/Fx	Merges injected code with the source file.
/I<dir>	Searches a directory for include files.
/P	Writes preprocessor output to a file.
`/PD`	Print all macro definitions.
`/PH`	Generate `#pragma file_hash` when preprocessing.

Option	Purpose
/U<name>	Removes a predefined macro.
/u	Removes all predefined macros.
/X	Ignores the standard include directory.

Header units/modules

Option	Purpose
/exportHeader	Create the header units files (`.ifc`) specified by the input arguments.
/headerUnit	Specify where to find the header unit file (`.ifc`) for the specified header.
/headerName	Build a header unit from the specified header.
/ifcOutput	Specify the output file name or directory for built `.ifc` files.
/interface	Treat the input file as a module interface unit.
/internalPartition	Treat the input file as an internal partition unit.
/reference	Use named module IFC.
/scanDependencies	List module and header unit dependencies in C++ Standard JSON form.
/sourceDependencies	List all source-level dependencies.
/sourceDependencies:directives	List module and header unit dependencies.
/translateInclude	Treat `#include` as `import`.

Language

Option	Purpose
/await	Enable coroutines (resumable functions) extensions.
/await:strict	Enable standard C++20 coroutine support with earlier language versions.
/constexpr:backtrace<N>	Show N `constexpr` evaluations in diagnostics (default: 10).

Option	Purpose
/constexpr:depth<N>	Recursion depth limit for `constexpr` evaluation (default: 512).
/constexpr:steps<N>	Terminate `constexpr` evaluation after N steps (default: 100000)
/openmp	Enables #pragma omp in source code.
/openmp:experimental	Enable OpenMP 2.0 language extensions plus select OpenMP 3.0+ language extensions.
/openmp:llvm	OpenMP language extensions using LLVM runtime.
/permissive[-]	Set standard-conformance mode.
/std:c++14	C++14 standard ISO/IEC 14882:2014 (default).
/std:c++17	C++17 standard ISO/IEC 14882:2017.
/std:c++20	C++20 standard ISO/IEC 14882:2020.
/std:c++latest	The latest draft C++ standard preview features.
/std:c11	C11 standard ISO/IEC 9899:2011.
/std:c17	C17 standard ISO/IEC 9899:2018.
/std:clatest	The latest draft C standard preview features.
/vd{0\|1\|2}	Suppresses or enables hidden `vtordisp` class members.
/vmb	Uses best base for pointers to members.
/vmg	Uses full generality for pointers to members.
/vmm	Declares multiple inheritance.
/vms	Declares single inheritance.
/vmv	Declares virtual inheritance.
/Z7	Generates C 7.0-compatible debugging information.
/Za	Disables some C89 language extensions in C code.
/Zc:__cplusplus[-]	Enable the `__cplusplus` macro to report the supported standard (off by default).
/Zc:__STDC__	Enable the `__STDC__` macro to report the C standard is supported (off by default).
/Zc:alignedNew[-]	Enable C++17 over-aligned dynamic allocation (on by default in C++17).

Option	Purpose
/Zc:auto[-]	Enforce the new Standard C++ meaning for `auto` (on by default).
/Zc:char8_t[-]	Enable or disable C++20 native `u8` literal support as `const char8_t` (off by default, except under `/std:c++20`).
/Zc:enumTypes[-]	Enable Standard C++ rules for inferred `enum` base types (Off b y default, not implied by `/permissive-`).
/Zc:externC[-]	Enforce Standard C++ rules for `extern "C"` functions (implied by `/permissive-`).
/Zc:externConstexpr[-]	Enable external linkage for `constexpr` variables (off by default).
/Zc:forScope[-]	Enforce Standard C++ `for` scoping rules (on by default).
/Zc:gotoScope	Enforce Standard C++ `goto` rules around local variable initialization (implied by `/permissive-`).
/Zc:hiddenFriend[-]	Enforce Standard C++ hidden friend rules (implied by `/permissive-`)
/Zc:implicitNoexcept[-]	Enable implicit `noexcept` on required functions (on by default).
/Zc:inline[-]	Remove unreferenced functions or data if they're COMDAT or have internal linkage only (off by default).
/Zc:lambda[-]	Enable new lambda processor for conformance-mode syntactic checks in generic lambdas.
/Zc:noexceptTypes[-]	Enforce C++17 `noexcept` rules (on by default in C++17 or later).
/Zc:nrvo[-]	Enable optional copy and move elisions (on by default under `/O2`, `/permissive-`, or `/std:c++20` or later).
/Zc:preprocessor[-]	Use the new conforming preprocessor (off by default, except in C11/C17).
/Zc:referenceBinding[-]	A UDT temporary won't bind to a non-const lvalue reference (off by default).
/Zc:rvalueCast[-]	Enforce Standard C++ explicit type conversion rules (off by default).
/Zc:sizedDealloc[-]	Enable C++14 global sized deallocation functions (on by default).
/Zc:strictStrings[-]	Disable string-literal to `char*` or `wchar_t*` conversion (off by default).
/Zc:templateScope[-]	Enforce Standard C++ template parameter shadowing rules (off by default).
/Zc:ternary[-]	Enforce conditional operator rules on operand types (off by default).

Option	Purpose
/Zc:threadSafeInit[-]	Enable thread-safe local static initialization (on by default).
/Zc:throwingNew[-]	Assume `operator new` throws on failure (off by default).
/Zc:tlsGuards[-]	Generate runtime checks for TLS variable initialization (on by default).
/Zc:trigraphs	Enable trigraphs (obsolete, off by default).
/Zc:twoPhase[-]	Use nonconforming template parsing behavior (conforming by default).
/Zc:wchar_t[-]	`wchar_t` is a native type, not a typedef (on by default).
/Zc:zeroSizeArrayNew[-]	Call member `new`/`delete` for 0-size arrays of objects (on by default).
/Ze	Deprecated. Enables C89 language extensions.
/Zf	Improves PDB generation time in parallel builds.
/ZH:[MD5\|SHA1\|SHA_256]	Specifies MD5, SHA-1, or SHA-256 for checksums in debug info.
/ZI	Includes debug information in a program database compatible with Edit and Continue. (x86 only)
/Zi	Generates complete debugging information.
/Zl	Removes the default library name from the *`.obj`* file.
/Zo[-]	Generate richer debugging information for optimized code.
/Zp[n]	Packs structure members.
/Zs	Checks syntax only.
/ZW	Produces an output file to run on the Windows Runtime.

Linking

Option	Purpose
/F	Sets stack size.
/LD	Creates a dynamic-link library.
/LDd	Creates a debug dynamic-link library.
/link	Passes the specified option to LINK.

Option	Purpose
/LN	Creates an MSIL `.netmodule`.
/MD	Compiles to create a multithreaded DLL, by using *MSVCRT.lib*.
/MDd	Compiles to create a debug multithreaded DLL, by using *MSVCRTD.lib*.
/MT	Compiles to create a multithreaded executable file, by using *LIBCMT.lib*.
/MTd	Compiles to create a debug multithreaded executable file, by using *LIBCMTD.lib*.

Miscellaneous

Option	Purpose
/?	Lists the compiler options.
@	Specifies a response file.
/analyze	Enables code analysis.
/bigobj	Increases the number of addressable sections in an .obj file.
/c	Compiles without linking.
/cgthreads	Specifies number of *cl.exe* threads to use for optimization and code generation.
/errorReport	Deprecated. Windows Error Reporting (WER) settings control error reporting.
/execution-charset	Set execution character set.
`/fastfail`	Enable fast-fail mode.
/FC	Displays the full path of source code files passed to *cl.exe* in diagnostic text.
/FS	Forces writes to the PDB file to be serialized through *MSPDBSRV.EXE*.
/H	Deprecated. Restricts the length of external (public) names.
/HELP	Lists the compiler options.
/J	Changes the default `char` type.
/JMC	Supports native C++ Just My Code debugging.
/kernel	The compiler and linker create a binary that can be executed in the Windows kernel.
/MP	Builds multiple source files concurrently.

Option	Purpose
/nologo	Suppresses display of sign-on banner.
`/presetPadding`	Zero initialize padding for stack based class types.
/showIncludes	Displays a list of all include files during compilation.
/source-charset	Set source character set.
/Tc	Specifies a C source file.
/TC	Specifies all source files are C.
/Tp	Specifies a C++ source file.
/TP	Specifies all source files are C++.
/utf-8	Set source and execution character sets to UTF-8.
/V	Deprecated. Sets the version string.
/validate-charset	Validate UTF-8 files for only compatible characters.
`/volatileMetadata`	Generate metadata on volatile memory accesses.
/Yc	Create *`.PCH`* file.
/Yd	Deprecated. Places complete debugging information in all object files. Use /Zi instead.
/Yl	Injects a PCH reference when creating a debug library.
/Yu	Uses a precompiled header file during build.
/Y-	Ignores all other precompiled-header compiler options in the current build.
/Zm	Specifies the precompiled header memory allocation limit.

Diagnostics

Option	Purpose
/diagnostics:caret[-]	Diagnostics format: prints column and the indicated line of source.
/diagnostics:classic	Use legacy diagnostics format.
/diagnostics	Diagnostics format: prints column information.
/external:anglebrackets	Treat all headers included via `<>` as external.

Option	Purpose
/external:env:<var>	Specify an environment variable with locations of external headers.
/external:I <path>	Specify location of external headers.
/external:templates[-]	Evaluate warning level across template instantiation chain.
/external:W<n>	Set warning level for external headers.
/options:strict	Unrecognized compiler options are errors.
/sdl	Enable more security features and warnings.
/w	Disable all warnings.
/W0, /W1, /W2, /W3, /W4	Set output warning level.
/w1<n>, /w2<n>, /w3<n>, /w4<n>	Set warning level for the specified warning.
/Wall	Enable all warnings, including warnings that are disabled by default.
/wd<n>	Disable the specified warning.
/we<n>	Treat the specified warning as an error.
/WL	Enable one-line diagnostics for error and warning messages when compiling C++ source code from the command line.
/wo<n>	Display the specified warning only once.
/Wv:xx[.yy[.zzzzz]]	Disable warnings introduced after the specified version of the compiler.
/WX	Treat warnings as errors.

Experimental options

Experimental options may only be supported by certain versions of the compiler. They may also behave differently in different compiler versions. Often the best, or only, documentation for experimental options is in the Microsoft C++ Team Blog.

Option	Purpose
/experimental:log	Enables experimental structured SARIF output.
/experimental:module	Enables experimental module support.

Deprecated and removed compiler options

Option	Purpose
/clr:noAssembly	Deprecated. Use /LN (Create MSIL Module) instead.
/errorReport	Deprecated. Error reporting is controlled by Windows Error Reporting (WER) settings.
/experimental:preprocessor	Deprecated. Enables experimental conforming preprocessor support. Use /Zc:preprocessor
/Fr	Deprecated. Creates a browse information file without local variables.
/Ge	Deprecated. Activates stack probes. On by default.
/Gm	Deprecated. Enables minimal rebuild.
/GX	Deprecated. Enables synchronous exception handling. Use /EH instead.
/GZ	Deprecated. Enables fast checks. Use /RTC1 instead.
/H	Deprecated. Restricts the length of external (public) names.
/Og	Deprecated. Uses global optimizations.
/QIfist	Deprecated. Once used to specify how to convert from a floating-point type to an integral type.
/V	Deprecated. Sets the `.obj` file version string.
/Wp64	Obsolete. Detects 64-bit portability problems.
/Yd	Deprecated. Places complete debugging information in all object files. Use /Zi instead.
/Zc:forScope-	Deprecated. Disables conformance in for loop scope.
/Ze	Deprecated. Enables language extensions.
/Zg	Removed in Visual Studio 2015. Generates function prototypes.

See also

C/C++ building reference
MSVC compiler options
MSVC compiler command-line syntax

Compiler options listed alphabetically

Article • 11/13/2023

This table contains an alphabetical list of compiler options. For a list of compiler options by category, see the Compiler options listed by category article.

Compiler options

Option	Purpose
@	Specifies a response file.
/?	Lists the compiler options.
/AI<dir>	Specifies a directory to search to resolve file references passed to the #using directive.
/analyze	Enables code analysis.
/arch:<IA32\|SSE\|SSE2\|AVX\|AVX2\|AVX512>	Minimum CPU architecture requirements. IA32, SSE, and SSE2 are x86 only.
`/arm64EC`	Generate code compatible with the arm64EC ABI.
/await	Enable coroutines (resumable functions) extensions.
/await:strict	Enable standard C++20 coroutine support with earlier language versions.
/bigobj	Increases the number of addressable sections in an .obj file.
/C	Preserves comments during preprocessing.
/c	Compiles without linking.
/cgthreads	Specifies number of *cl.exe* threads to use for optimization and code generation.
/clr	Produces an output file to run on the common language runtime.
/clr:implicitKeepAlive-	Turn off implicit emission of `System::GC::KeepAlive(this)`.
/clr:initialAppDomain	Enable initial AppDomain behavior of Visual C++ 2002.
/clr:netcore	Produce assemblies targeting .NET Core runtime.

Option	Purpose
/clr:noAssembly	Don't produce an assembly.
/clr:nostdimport	Don't import any required assemblies implicitly.
/clr:nostdlib	Ignore the system .NET framework directory when searching for assemblies.
/clr:pure	Produce an IL-only output file (no native executable code).
/clr:safe	Produce an IL-only verifiable output file.
/constexpr:backtrace<N>	Show N `constexpr` evaluations in diagnostics (default: 10).
/constexpr:depth<N>	Recursion depth limit for `constexpr` evaluation (default: 512).
/constexpr:steps<N>	Terminate `constexpr` evaluation after N steps (default: 100000)
/D<name>{=\|#}<text>	Defines constants and macros.
/diagnostics	Diagnostics format: prints column information.
/diagnostics:caret[-]	Diagnostics format: prints column and the indicated line of source.
/diagnostics:classic	Use legacy diagnostics format.
/doc	Processes documentation comments to an XML file.
/E	Copies preprocessor output to standard output.
/EHa	Enable C++ exception handling (with SEH exceptions).
/EHc	`extern "C"` defaults to `nothrow`.
/EHr	Always generate `noexcept` runtime termination checks.
/EHs	Enable C++ exception handling (no SEH exceptions).
/EP	Copies preprocessor output to standard output.
/errorReport	Deprecated. Windows Error Reporting (WER) settings control error reporting.
/execution-charset	Set execution character set.
/experimental:log	Enables experimental structured SARIF output.
/experimental:module	Enables experimental module support.

Option	Purpose
/exportHeader	Create the header units files (`.ifc`) specified by the input arguments.
/external:anglebrackets	Treat all headers included via `<>` as external.
/external:env:<var>	Specify an environment variable with locations of external headers.
/external:I <path>	Specify location of external headers.
/external:templates[-]	Evaluate warning level across template instantiation chain.
/external:W<n>	Set warning level for external headers.
/F	Sets stack size.
/FA	Configures an assembly listing file.
/Fa	Creates an assembly listing file.
`/fastfail`	Enable fast-fail mode.
/favor:<blend\|AMD64\|INTEL64\|ATOM>	Produces code that is optimized for a specified architecture, or for a range of architectures.
/FC	Displays the full path of source code files passed to *cl.exe* in diagnostic text.
/Fd	Renames program database file.
/Fe	Renames the executable file.
/FI<file>	Preprocesses the specified include file.
/Fi	Specifies the preprocessed output file name.
/Fm	Creates a mapfile.
/Fo	Creates an object file.
/Fp	Specifies a precompiled header file name.
/fp:contract	Consider floating-point contractions when generating code.
/fp:except[-]	Consider floating-point exceptions when generating code.
/fp:fast	"fast" floating-point model; results are less predictable.
/fp:precise	"precise" floating-point model; results are predictable.

Option	Purpose
/fp:strict	"strict" floating-point model (implies `/fp:except`).
/fpcvt:BC	Backward-compatible floating-point to unsigned integer conversions.
/fpcvt:IA	Intel native floating-point to unsigned integer conversion behavior.
/FR, /Fr	Name generated `.sbr` browser files. `/Fr` is deprecated.
/FS	Forces writes to the PDB file to be serialized through *MSPDBSRV.EXE*.
/fsanitize	Enables compilation of sanitizer instrumentation such as AddressSanitizer.
/fsanitize-coverage	Enables compilation of code coverage instrumentation for libraries such as LibFuzzer.
`/Ft<dir>`	Location of the header files generated for `#import`.
/FU<file>	Forces the use of a file name, as if it had been passed to the #using directive.
/Fx	Merges injected code with the source file.
/GA	Optimizes for Windows applications.
/Gd	Uses the `__cdecl` calling convention. (x86 only)
/Ge	Deprecated. Activates stack probes.
/GF	Enables string pooling.
/GH	Calls hook function `_pexit`.
/Gh	Calls hook function `_penter`.
/GL[-]	Enables whole program optimization.
/Gm[-]	Deprecated. Enables minimal rebuild.
/GR[-]	Enables run-time type information (RTTI).
/Gr	Uses the `__fastcall` calling convention. (x86 only)
/GS[-]	Checks buffer security.
/Gs[n]	Controls stack probes.

Option	Purpose
/GT	Supports fiber safety for data allocated by using static thread-local storage.
`/Gu[-]`	Ensure distinct functions have distinct addresses.
/guard:cf[-]	Adds control flow guard security checks.
/guard:ehcont[-]	Enables EH continuation metadata.
/Gv	Uses the `__vectorcall` calling convention. (x86 and x64 only)
/Gw[-]	Enables whole-program global data optimization.
/GX[-]	Deprecated. Enables synchronous exception handling. Use /EH instead.
/Gy[-]	Enables function-level linking.
/GZ	Deprecated. Enables fast checks. (Same as /RTC1)
/Gz	Uses the `__stdcall` calling convention. (x86 only)
/H	Deprecated. Restricts the length of external (public) names.
/headerName	Build a header unit from the specified header.
/headerUnit	Specify where to find the header unit file (`.ifc`) for the specified header.
/HELP	Lists the compiler options.
/homeparams	Forces parameters passed in registers to be written to their locations on the stack upon function entry. This compiler option is only for the x64 compilers (native and cross compile).
/hotpatch	Creates a hotpatchable image.
/I<dir>	Searches a directory for include files.
/ifcOutput	Specify output file name or directory for built `.ifc` files.
/interface	Treat the input file as a module interface unit.
/internalPartition	Treat the input file as an internal partition unit.
/J	Changes the default `char` type.

Option	Purpose
/jumptablerdata	Put switch case statement jump tables in the `.rdata` section.
/JMC	Supports native C++ Just My Code debugging.
/kernel	The compiler and linker create a binary that can be executed in the Windows kernel.
/LD	Creates a dynamic-link library.
/LDd	Creates a debug dynamic-link library.
/link	Passes the specified option to LINK.
/LN	Creates an MSIL `.netmodule`.
/MD	Compiles to create a multithreaded DLL, by using *MSVCRT.lib*.
/MDd	Compiles to create a debug multithreaded DLL, by using *MSVCRTD.lib*.
/MP	Builds multiple source files concurrently.
/MT	Compiles to create a multithreaded executable file, by using *LIBCMT.lib*.
/MTd	Compiles to create a debug multithreaded executable file, by using *LIBCMTD.lib*.
/nologo	Suppresses display of sign-on banner.
/O1	Creates small code.
/O2	Creates fast code.
/Ob<n>	Controls inline expansion.
/Od	Disables optimization.
/Og	Deprecated. Uses global optimizations.
/Oi[-]	Generates intrinsic functions.
/openmp	Enables #pragma omp in source code.
/openmp:experimental	Enable OpenMP 2.0 language extensions plus select OpenMP 3.0+ language extensions.
/openmp:llvm	OpenMP language extensions using LLVM runtime.

Option	Purpose
/options:strict	Unrecognized compiler options are errors.
/Os	Favors small code.
/Ot	Favors fast code.
/Ox	A subset of /O2 that doesn't include /GF or /Gy.
/Oy	Omits frame pointer. (x86 only)
/P	Writes preprocessor output to a file.
`/PD`	Print all macro definitions.
/permissive[-]	Set standard-conformance mode.
`/PH`	Generate `#pragma file_hash` when preprocessing.
`/presetPadding`	Zero initialize padding for stack based class types.
/Qfast_transcendentals	Generates fast transcendentals.
/QIfist	Deprecated. Suppresses the call of the helper function `_ftol` when a conversion from a floating-point type to an integral type is required. (x86 only)
/Qimprecise_fwaits	Removes `fwait` commands inside `try` blocks.
/QIntel-jcc-erratum	Mitigates the performance impact of the Intel JCC erratum microcode update.
/Qpar-report:<n>	Enables reporting levels for automatic parallelization.
/Qpar	Enables automatic parallelization of loops.
/Qsafe_fp_loads	Uses integer move instructions for floating-point values and disables certain floating point load optimizations.
/Qspectre[-]	Enable mitigations for CVE 2017-5753, for a class of Spectre attacks.
/Qspectre-load	Generate serializing instructions for every load instruction.
/Qspectre-load-cf	Generate serializing instructions for every control flow instruction that loads memory.
/Qvec-report:<n>	Enables reporting levels for automatic vectorization.
/reference	Use named module IFC.
/RTC1	Enable fast runtime checks (equivalent to `/RTCsu`).

Option	Purpose
/RTCc	Convert to smaller type checks at run-time.
/RTCs	Enable stack frame runtime checks.
/RTCu	Enables uninitialized local usage checks.
/scanDependencies	List module dependencies in C++ Standard JSON form.
/sdl	Enable more security features and warnings.
/showIncludes	Displays a list of all include files during compilation.
/source-charset	Set source character set.
/sourceDependencies	List all source-level dependencies.
/sourceDependencies:directives	List module and header unit dependencies.
/std:c++14	C++14 standard ISO/IEC 14882:2014 (default).
/std:c++17	C++17 standard ISO/IEC 14882:2017.
/std:c++20	C++20 standard ISO/IEC 14882:2020.
/std:c++latest	The latest draft C++ standard preview features.
/std:c11	C11 standard ISO/IEC 9899:2011.
/std:c17	C17 standard ISO/IEC 9899:2018.
/std:clatest	The latest draft C standard preview features.
/TC	Specifies all source files are C.
/Tc	Specifies a C source file.
/TP	Specifies all source files are C++.
/Tp	Specifies a C++ source file.
/translateInclude	Treat `#include` as `import`.
/U<name>	Removes a predefined macro.
/u	Removes all predefined macros.
/utf-8	Set source and execution character sets to UTF-8.
/V	Deprecated. Sets the version string.
/validate-charset	Validate UTF-8 files for only compatible characters.

Option	Purpose
/vd{0\|1\|2}	Suppresses or enables hidden `vtordisp` class members.
/vmb	Uses best base for pointers to members.
/vmg	Uses full generality for pointers to members.
/vmm	Declares multiple inheritance.
/vms	Declares single inheritance.
/vmv	Declares virtual inheritance.
/volatile:iso	Acquire/release semantics not guaranteed on volatile accesses.
/volatile:ms	Acquire/release semantics guaranteed on volatile accesses.
`/volatileMetadata`	Generate metadata on volatile memory accesses.
/w	Disable all warnings.
/W0, /W1, /W2, /W3, /W4	Set output warning level.
/w1<n>, /w2<n>, /w3<n>, /w4<n>	Set warning level for the specified warning.
/Wall	Enable all warnings, including warnings that are disabled by default.
/wd<n>	Disable the specified warning.
/we<n>	Treat the specified warning as an error.
/WL	Enable one-line diagnostics for error and warning messages when compiling C++ source code from the command line.
/wo<n>	Display the specified warning only once.
/Wv:xx[.yy[.zzzzz]]	Disable warnings introduced after the specified version of the compiler.
/WX	Treat warnings as errors.
/X	Ignores the standard include directory.
/Y-	Ignores all other precompiled-header compiler options in the current build.
/Yc	Create *`.PCH`* file.

Option	Purpose
/Yd	Deprecated. Places complete debugging information in all object files. Use /Zi instead.
/Yl	Injects a PCH reference when creating a debug library.
/Yu	Uses a precompiled header file during build.
/Z7	Generates C 7.0-compatible debugging information.
/Za	Disables some C89 language extensions in C code.
/Zc:__cplusplus[-]	Enable the `__cplusplus` macro to report the supported standard (off by default).
/Zc:__STDC__	Enable the `__STDC__` macro to report the C standard is supported (off by default).
/Zc:alignedNew[-]	Enable C++17 over-aligned dynamic allocation (on by default in C++17).
/Zc:auto[-]	Enforce the new Standard C++ meaning for **`auto`** (on by default).
/Zc:char8_t[-]	Enable or disable C++20 native `u8` literal support as `const char8_t` (off by default, except under **`/std:c++20`**).
/Zc:enumTypes[-]	Enable Standard C++ rules for `enum` type deduction (off by default).
/Zc:externC[-]	Enforce Standard C++ rules for `extern "C"` functions (implied by **`/permissive-`**).
/Zc:externConstexpr[-]	Enable external linkage for **`constexpr`** variables (off by default).
/Zc:forScope[-]	Enforce Standard C++ **`for`** scoping rules (on by default).
/Zc:gotoScope	Enforce Standard C++ **`goto`** rules around local variable initialization (implied by **`/permissive-`**).
/Zc:hiddenFriend[-]	Enforce Standard C++ hidden friend rules (implied by **`/permissive-`**)
/Zc:implicitNoexcept[-]	Enable implicit **`noexcept`** on required functions (on by default).
/Zc:inline[-]	Remove unreferenced functions or data if they're COMDAT or have internal linkage only (off by default).

Option	Purpose
/Zc:lambda[-]	Enable new lambda processor for conformance-mode syntactic checks in generic lambdas.
/Zc:noexceptTypes[-]	Enforce C++17 `noexcept` rules (on by default in C++17 or later).
/Zc:nrvo[-]	Enable optional copy and move elisions (on by default under `/O2`, `/permissive-`, or `/std:c++20` or later).
/Zc:preprocessor[-]	Use the new conforming preprocessor (off by default, except in C11/C17).
/Zc:referenceBinding[-]	A UDT temporary won't bind to a non-const lvalue reference (off by default).
/Zc:rvalueCast[-]	Enforce Standard C++ explicit type conversion rules (off by default).
/Zc:sizedDealloc[-]	Enable C++14 global sized deallocation functions (on by default).
/Zc:strictStrings[-]	Disable string-literal to `char*` or `wchar_t*` conversion (off by default).
/Zc:templateScope[-]	Enforce Standard C++ template parameter shadowing rules (off by default).
/Zc:ternary[-]	Enforce conditional operator rules on operand types (off by default).
/Zc:threadSafeInit[-]	Enable thread-safe local static initialization (on by default).
/Zc:throwingNew[-]	Assume `operator new` throws on failure (off by default).
/Zc:tlsGuards[-]	Generate runtime checks for TLS variable initialization (on by default).
/Zc:trigraphs	Enable trigraphs (obsolete, off by default).
/Zc:twoPhase[-]	Use nonconforming template parsing behavior (conforming by default).
/Zc:wchar_t[-]	`wchar_t` is a native type, not a typedef (on by default).
/Zc:zeroSizeArrayNew[-]	Call member `new`/`delete` for zero-size arrays of objects (on by default).
/Ze	Deprecated. Enables C89 language extensions.
/Zf	Improves PDB generation time in parallel builds.

Option	Purpose
/ZH:[MD5\|SHA1\|SHA_256]	Specifies MD5, SHA-1, or SHA-256 for checksums in debug info.
/ZI	Includes debug information in a program database compatible with Edit and Continue. (x86 only)
/Zi	Generates complete debugging information.
/Zl	Removes the default library name from the `.obj` file.
/Zm	Specifies the precompiled header memory allocation limit.
/Zo[-]	Generate richer debugging information for optimized code.
/Zp[n]	Packs structure members.
/Zs	Checks syntax only.
/ZW	Produces an output file to run on the Windows Runtime.

See also

MSVC compiler options
MSVC compiler command-line syntax

@ (Specify a Compiler Response File)

Article • 08/03/2021

Specifies a compiler response file.

Syntax

> @*response_file*

Arguments

response_file
A text file containing compiler commands.

Remarks

A response file can contain any commands that you would specify on the command line. This can be useful if your command-line arguments exceed 127 characters.

It is not possible to specify the @ option from within a response file. That is, a response file cannot embed another response file.

From the command line you can specify as many response file options (for example, `@respfile.1 @respfile.2`) as you want.

To set this compiler option in the Visual Studio development environment

- A response file cannot be specified from within the development environment and must be specified at the command line.

To set this compiler option programmatically

- This compiler option cannot be changed programmatically.

See also

MSVC Compiler Options

MSVC Compiler Command-Line Syntax

/AI (Specify Metadata Directories)

Article • 08/03/2021

Specifies a directory that the compiler will search to resolve file references passed to the `#using` directive.

Syntax

> **/AI***directory*

Arguments

directory
The directory or path for the compiler to search.

Remarks

Only one directory can be passed to an **/AI** invocation. Specify one **/AI** option for each path you want the compiler to search. For example, to add both C:\Project\Meta and C:\Common\Meta to the compiler search path for `#using` directives, add `/AI"C:\Project\Meta" /AI"C:\Common\Meta"` to the compiler command line or add each directory to the **Additional #using Directories** property in Visual Studio.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **General** property page.
3. Modify the **Additional #using Directories** property.

To set this compiler option programmatically

- See AdditionalUsingDirectories.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
#using Directive

`/analyze` (Code analysis)

Article • 02/18/2022

Enables code analysis and control options.

Syntax

General analysis options:

> **`/analyze`**[**`-`**]
> **`/analyze:only`**
> **`/analyze:quiet`**
> **`/analyze:max_paths`** *`number`*
> **`/analyze:stacksize`** *`number`*
> **`/analyze:WX-`**

Analysis plugin options:

> **`/analyze:plugin`** *`plugin_dll`*

External file analysis options:

> **`/analyze:external-`**
> **`/analyze:external:ruleset`** *`ruleset_files`*

Analysis log options:

> **`/analyze:autolog`**[**`-`**]
> **`/analyze:autolog:ext`** *`extension`*
> **`/analyze:log`** *`log_path`*

Log file format options:

> **`/analyze:log:format:sarif`**
> **`/analyze:log:format:xml`**

Log file content options:

> **`/analyze:sarif:analyzedfiles`**[**`-`**]
> **`/analyze:sarif:configuration`**[**`-`**]
> **`/analyze:log:compilerwarnings`**
> **`/analyze:log:includesuppressed`**

Ruleset options:

> **`/analyze:projectdirectory`** *`project_directory`*
> **`/analyze:rulesetdirectory`** *`ruleset_directories`*
> **`/analyze:ruleset`** *`ruleset_files`*

Arguments

General analysis options

`/analyze`[**`-`**]
Turns on code analysis. Use **`/analyze-`** to explicitly turn off analysis. **`/analyze-`** is the default behavior.

By default, analysis output goes to the console or the Visual Studio Output window like other error messages. Code analysis also creates a log file named *`filename.nativecodeanalysis.xml`*, where *`filename`* is the name of the analyzed source file.

`/analyze:only`
By default, the compiler compiles the code to generate object files before code analysis runs. The **`/analyze:only`** option makes the compiler skip the code generation pass, and does code analysis directly. Compiler errors still prevent code analysis from running. However, the compiler won't report other warnings that it might find during the code generation pass. If the program isn't free of code-generation warnings, analysis results might be unreliable. We recommend you use this option only if the code passes code-generation syntax checks without errors or warnings.

`/analyze:quiet`
Turns off analysis output to the console or Visual Studio Output window.

`/analyze:max_paths` *`number`*
The *`number`* parameter specifies the maximum number of code paths to analyze. Analysis defaults to 256 paths. Larger values cause more thorough checking, but the analysis might take longer.

`/analyze:stacksize` *`number`*

The *`number`* parameter specifies the size in bytes of the stack frame that generates warning C6262. The default stack frame size is 16KB.

`/analyze:WX-`

Tells the compiler not to treat code analysis warnings as errors even when the **`/WX`** option is used. For more information, see /WX (Warning level).

Analysis plugin options

`/analyze:plugin` *`plugin_dll`*

Enables the specified code analysis plug-in DLL for code analysis.

Space between **`/analyze:plugin`** and the *`plugin_dll`* file path is optional if the path doesn't require double-quotes (`"`). For example, you can write `/analyze:plugin EspxEngine.dll`. However, if the path is enclosed in double-quotes, you can't have a space between **`/analyze:plugin`** and the file path. Here's an example: `/analyze:plugin"c:\path\to\EspxEngine.dll"`.

The code analysis engine uses plug-ins to help find specific categories of defects. The code analysis engine comes with some built-in plug-ins that detect various defects. To use another plug-in with the code analysis engine, specify it by using the **`/analyze:plugin`** option.

Some plug-ins, like `EspXEngine.dll`, which ships with Visual Studio, employ extensions that can do further analysis. Visual Studio includes these extensions for EspXEngine: `ConcurrencyCheck.dll`, `CppCoreCheck.dll`, `EnumIndex.dll`, `HResultCheck.dll`, and `VariantClear.dll`. They check for defects for concurrency issues, CppCoreGuidelines violations, inappropriate uses of `enum` values as indexes, `HRESULT` values, or `VARIANT` values, respectively.

When you build on the command line, you can use the `Esp.Extensions` environment variable to specify EspXEngine extensions. For example:

Windows Command Prompt

```
set Esp.Extensions=ConcurrencyCheck.dll;CppCoreCheck.dll;
```

Use a semicolon (`;`) to delimit the extensions, as shown in the example. A trailing semicolon isn't needed. You can use an absolute file path for an extension, or specify a relative path from the directory that contains `EspXEngine.dll`.

The `EspXEngine.dll` plug-in uses `ConcurrencyCheck.dll` to implement concurrency-related code analysis checks. These checks raise warnings in the C261XX range, such as C26100 through C26167.

If you're building in a developer command prompt window, first set the `Esp.Extensions` environment variable to specify the `ConcurrencyCheck.dll` extension:

Windows Command Prompt

```
set Esp.Extensions=ConcurrencyCheck.dll
```

Then, use compiler option `/analyze:plugin EspXEngine.dll` to use the EspXEngine plug-in.

External file analysis options

Starting in Visual Studio 2019 version 16.10, you can specify different analysis rules and behavior for external headers. Use the **`/external:I`**, **`/external:env`**, or **`/external:anglebrackets`** options to specify directories as "external" directories. Any files that are included by using `#include` from an external directory or its subdirectories are considered as external headers. For more information, see /external (External headers diagnostics).

Code analysis provides these options to control analysis of external files:

`/analyze:external-`
Skips analysis of external header files. By default, code analysis analyzes external header files just like other files. When the **`/analyze:external-`** option is set, code analysis skips any files specified as external, except templates from external files. Templates defined in external headers are treated as non-external by using the **`/external:templates-`** option. The **`/external:Wn`** option doesn't affect code analysis. For example, code analysis analyzes external files and reports defects even when `/external:W0` is specified.

`/analyze:external:ruleset` *`ruleset_files`*
The *`ruleset_files`* parameter specifies one or more semicolon-delimited ruleset files to use for analysis of external files. For information on rulesets, refer to "Options for rulesets" section.

There's an environment variable (`CAExcludePath`) that provides similar but simpler capability to skip analysis of files under the directories specified in the environment variable. If a directory is specified in both **`/external:*`** option and in the `CAExcludePath`

environment variable, it's considered as excluded, and `/analyze:external*` options won't apply to that directory.

Analysis log options

`/analyze:autolog`[`-`]
This flag used to be required to enable creation of analysis log file for each of the source files being analyzed. Log files are now created by default, so this flag is mostly redundant. When used, it changes the default log extension to `*.pftlog` instead of `.xml`. Use `/analyze:autolog-` to disable logging to files.

`/analyze:autolog:ext` *`extension`*
Overrides the default extension of the analysis log files, and uses *`extension`* instead. If you use the `.sarif` extension, the log file uses the SARIF format instead of the default XML format.

`/analyze:log` *`log_path`*
Specifies a log file path *`log_path`* instead of the automatically generated log file path. When the *`log_path`* path has a trailing backslash and refers to an existing directory, code analysis creates all log files in the specified directory. Otherwise, *`log_path`* specifies a file path. A file path instructs the compiler to combine logs for all analyzed source files into the specified log file. If the file path has a `.sarif` extension, the log file uses the SARIF format instead of the default XML format. You can override this behavior by using the `/analyze:log:format:*` option.

Log file format options

Starting in Visual Studio 2019 version 16.9, you can specify different log format options for code analysis.

`/analyze:log:format:xml`
Forces the use of XML log format irrelevant of the file extension used.

`/analyze:log:format:sarif`
Forces the use of SARIF log format irrelevant of the file extension used.

Log file content options

Starting in Visual Studio 2019 version 16.9, you can specify different log content options for code analysis.

`/analyze:sarif:analyzedfiles`[**`-`**]
Adds file artifacts entries to the SARIF log file for analyzed files that don't issue warnings. This option is disabled by default. Artifacts for the source file and for files that emitted results are always included.

`/analyze:sarif:configuration`[**`-`**]
Adds rule configuration entries to determine how the user overrode the default rule configuration (disabled by default).

`/analyze:log:compilerwarnings`
Adds both any defects the analysis engine finds, and all compiler warnings, to the analysis log file. By default, compiler warnings aren't included in the analysis log file. For more information on compiler warnings during code analysis, see the **`/analyze:only`** option.

`/analyze:log:includesuppressed`
Adds both suppressed warnings and unsuppressed warnings to the analysis log file. By default, suppressed warnings aren't included in the analysis log file. If ruleset files are specified for analysis, the warnings disabled by the ruleset files aren't included in the log even when **`/analyze:log:includesuppressed`** is specified.

Ruleset options

`/analyze:projectdirectory` *`project_directory`*
Specifies the current project directory. If the ruleset (or an item it includes) is a file name, the compiler first looks for the file under the specified *`project_directory`*. If not found, it next searches the *`ruleset_directories`* specified by **`/analyze:rulesetdirectory`**, if any. If the ruleset (or an item it includes) is a relative path, the compiler first looks for the file under the project directory. If the ruleset isn't found, then it looks in the current working directory. This option is available starting in Visual Studio 2019 version 16.9.

`/analyze:rulesetdirectory` *`ruleset_directories`*
Specifies a semicolon-separated list of ruleset search paths. If the ruleset (or an item it includes) is a file name, then the compiler first looks for the file under the *`project_directory`* specified by **`/analyze:projectdirectory`**, if any, followed by the specified *`ruleset_directories`*. This option is available starting in Visual Studio 2019 version 16.9.

`/analyze:ruleset` *`ruleset_files`*
Specifies one or more ruleset files to use for analysis. This option can make analysis more efficient; the analysis engine tries to exclude checkers that have no active rules

specified in the ruleset files before running. Otherwise, the engine runs all checkers enabled.

The ruleset files that ship with Visual Studio are found in `%VSINSTALLDIR%\Team Tools\Static Analysis Tools\Rule Sets`.

The following example custom ruleset tells the analysis engine to check for C6001 and C26494, and report them as warnings.

You can place this file anywhere as long as you specify the full path in the argument, or under the directories specified in the **`/analyze:projectdirectory`** or **`/analyze:rulesetdirectory`** options.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="New Rule Set" Description="New rules to apply."
ToolsVersion="15.0">
  <Rules AnalyzerId="Microsoft.Analyzers.NativeCodeAnalysis"
RuleNamespace="Microsoft.Rules.Native">
    <Rule Id="C6001" Action="Warning" />
    <Rule Id="C26494" Action="Warning" />
  </Rules>
</RuleSet>
```

By default, the file extension for ruleset files is `*.ruleset`. Visual Studio uses the default extension when browsing for ruleset files. However, you can use any extension.

For more information about rulesets, see Use rule sets to specify the C++ rules to run.

Remarks

For more information, see Code analysis for C/C++ overview and Code analysis for C/C++ warnings.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Code Analysis** > **General** property page.
3. Modify one or more of the **Code Analysis** properties.

4. Choose **OK** or **Apply** to save your changes.

To set external file analysis options in Visual Studio 2019 version 16.10 and later:

1. Open the project's **Property Pages** dialog box.
2. Select the **Configuration Properties** > **C/C++** > **External Includes** property page.
3. Set properties:
 - **Disable Code Analysis for External Headers** sets the `/analyze:external-` option.
 - **Analysis Ruleset for External Headers** sets the `/analyze:external:ruleset path` option.
4. Choose **OK** or **Apply** to save your changes.

To set this compiler option programmatically

1. See EnablePREfast.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/arch` (Minimum CPU Architecture)

Article • 07/01/2022

The architecture options specify the architecture for code generation. Select the base hardware architecture you're working with to see `/arch` options for that target platform.

- /arch (x86)
- /arch (x64)
- /arch (ARM)
- /arch (ARM64)

See also

MSVC compiler options
MSVC compiler command-line syntax

`/arch` (x86)

Article • 10/16/2024

Specifies the architecture for code generation on x86. For more information on `/arch` for other target architectures, see /arch (ARM64), /arch (x64), and /arch (ARM).

Syntax

> `/arch:`[`IA32`|`SSE`|`SSE2`|`AVX`|`AVX2`|`AVX512`|`AVX10.1`]

Arguments

`/arch:IA32`
Specifies no enhanced instructions and also specifies x87 for floating-point calculations.

`/arch:SSE`
Enables Intel Streaming SIMD Extensions.

`/arch:SSE2`
Enables Intel Streaming SIMD Extensions 2. The default instruction set is SSE2 if no `/arch` option is specified.

`/arch:AVX`
Enables Intel Advanced Vector Extensions.

`/arch:AVX2`
Enables Intel Advanced Vector Extensions 2.

`/arch:AVX512`
Enables Intel Advanced Vector Extensions 512.

`/arch:AVX10.1`
Enables Intel Advanced Vector Extensions 10 version 1.

Remarks

The `/arch` option enables or disables the use of certain instruction set extensions, particularly for vector calculation, available in processors from Intel and AMD. In general, more recently introduced processors may support extensions beyond the ones supported by older processors. You should consult the documentation for a particular

processor or test for instruction set extension support using __cpuid before executing code using an instruction set extension.

`/arch` only affects code generation for native functions. When you use /clr to compile, `/arch` has no effect on code generation for managed functions.

The `/arch` options refer to instruction set extensions with the following characteristics:

- `IA32` is the legacy 32-bit x86 instruction set without any vector operations and using x87 for floating-point calculations.
- `SSE` allows calculation with vectors of up to four single-precision floating-point values. Corresponding scalar floating-point instructions were added as well.
- `SSE2` allows calculation with 128-bit vectors of single-precision, double-precision and 1, 2, 4, or 8-byte integer values. Double-precision scalar instructions were also added.
- `AVX` introduced an alternative instruction encoding for vector and floating-point scalar instructions. It allows vectors of either 128 bits or 256 bits, and zero-extends all vector results to the full vector size. (For legacy compatibility, SSE-style vector instructions preserve all bits beyond bit 127.) Most floating-point operations are extended to 256 bits.
- `AVX2` extends most integer operations to 256-bit vectors, and enables use of Fused Multiply-Add (FMA) instructions.
- `AVX512` introduced another instruction encoding form that allows 512-bit vectors, masking, embedded rounding/broadcast, and new instructions. The default vector length for `AVX512` is 512 bits and can be changed to 256 bits using the /vlen flag.
- `AVX10.1` adds more instructions on top of `AVX-512`. The default vector length for `AVX10.1` is 256 bits and can be changed to 512 bits using the /vlen flag.

The optimizer chooses when and how to use vector instructions depending on which `/arch` is specified. Scalar floating-point computations are usually performed with SSE or AVX instructions when available. Some calling conventions specify passing floating-point arguments on the x87 stack, and as a result, your code may use a mixture of both x87 and SSE/AVX instructions for floating-point computations. Integer vector instructions can also be used for some 64-bit integer operations when available.

In addition to the vector and floating-point scalar instructions, each `/arch` option may also enable the use of other non-vector instructions that are associated with that option. An example is the CMOVcc instruction family that first appeared on the Intel Pentium

Pro processors. Because SSE instructions were introduced with the subsequent Intel Pentium III processor, CMOVcc instructions may be generated except when `/arch:IA32` is specified.

Floating-point operations are normally rounded to double-precision (64-bit) in x87 code, but you can use `_controlfp` to modify the FP control word, including setting the precision control to extended precision (80-bit) or single-precision (32-bit). For more information, see _control87, _controlfp, __control87_2. SSE and AVX have separate single-precision and double-precision instructions for each operation, so there's no equivalent for SSE/AVX code. It can change how results are rounded when the result of a floating-point operation is used directly in further calculation instead of assigning it to a user variable. Consider the following operations:

C++

```
r = f1 * f2 + d;  // Different results are possible on SSE/SSE2.
```

With explicit assignment:

C++

```
t = f1 * f2;    // Do f1 * f2, round to the type of t.
r = t + d;      // This should produce the same overall result
                // whether x87 stack is used or SSE/SSE2 is used.
```

`/arch` and /QIfist can't be used together. The `/QIfist` option changes the rounding behavior of floating-point to integer conversion. The default behavior is to truncate (round toward zero), whereas the `/QIfist` option specifies use of the floating-point environment rounding mode. Because the option changes the behavior of all floating-point to integer conversions, `/QIfist` is deprecated. When compiling for SSE or AVX, you can round a floating-point value to an integer using the floating-point environment rounding mode by using an intrinsic function sequence:

C++

```
int convert_float_to_int(float x) {
    return _mm_cvtss_si32(_mm_set_ss(x));
}

int convert_double_to_int(double x) {
    return _mm_cvtsd_si32(_mm_set_sd(x));
}
```

The `_M_IX86_FP`, `__AVX__`, `__AVX2__`, `__AVX512F__`, `__AVX512CD__`, `__AVX512BW__`, `__AVX512DQ__`, `__AVX512VL__`, and `__AVX10_VER__` macros indicate which, if any, **`/arch`** compiler option was used. For more information, see Predefined macros. The **`/arch:AVX2`** option, and `__AVX2__` macro were introduced in Visual Studio 2013 Update 2, version 12.0.34567.1. Limited support for **`/arch:AVX512`** was added in Visual Studio 2017, and expanded in Visual Studio 2019. Support for **`/arch:AVX10.1`** was added in Visual Studio 2022.

To set the `/arch` compiler option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify the **Enable Enhanced Instruction Set** property.

To set this compiler option programmatically

- See EnableEnhancedInstructionSet.

See also

/arch (Minimum CPU Architecture)
MSVC compiler options
MSVC compiler command-line syntax

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/arch` (x64)

Article • 10/16/2024

Specifies the architecture for code generation on x64. For more information on `/arch` for other target architectures, see /arch (x86), /arch (ARM64), and /arch (ARM).

Syntax

> `/arch:`[`SSE2`|`SSE4.2`|`AVX`|`AVX2`|`AVX512`|`AVX10.1`]

Arguments

`/arch:SSE2`
Enables Intel Streaming SIMD Extensions 2. The default instruction set is SSE2 if no `/arch` option is specified.

`/arch:SSE4.2`
Enables Intel Streaming SIMD Extensions 4.2.

`/arch:AVX`
Enables Intel Advanced Vector Extensions.

`/arch:AVX2`
Enables Intel Advanced Vector Extensions 2.

`/arch:AVX512`
Enables Intel Advanced Vector Extensions 512.

`/arch:AVX10.1`
Enables Intel Advanced Vector Extensions 10 version 1.

Remarks

The `/arch` option enables the use of certain instruction set extensions, particularly for vector calculation, available in processors from Intel and AMD. In general, more recently introduced processors may support extensions beyond the ones supported by older processors, although you should consult the documentation for a particular processor or test for instruction set extension support using __cpuid before executing code using an instruction set extension.

`/arch` only affects code generation for native functions. When you use /clr to compile, `/arch` has no effect on code generation for managed functions.

The processor extensions have the following characteristics:

- The default mode uses SSE2 instructions for scalar floating-point and vector calculations. These instructions allow calculation with 128-bit vectors of single-precision, double-precision and 1, 2, 4 or 8-byte integer values, as well as single-precision and double-precision scalar floating-point values.
- `SSE4.2` uses the full set of SSE instructions for floating-point scalar, vector, and integer vector calculations.
- `AVX` introduced an alternative instruction encoding for vector and floating-point scalar instructions. It allows vectors of either 128 bits or 256 bits, and zero-extends all vector results to the full vector size. (For legacy compatibility, SSE-style vector instructions preserve all bits beyond bit 127.) Most floating-point operations are extended to 256 bits.
- `AVX2` extends most integer operations to 256-bit vectors and enables use of Fused Multiply-Add (FMA) instructions.
- `AVX-512` introduced another instruction encoding form that allows 512-bit vectors, masking, embedded rounding/broadcast, and new instructions. The default vector length for `AVX-512` is 512 bits and can be changed to 256 bits using the /vlen flag.
- `AVX10.1` adds more instructions on top of `AVX-512`. The default vector length for `AVX10.1` is 256 bits and can be changed to 512 bits using the /vlen flag.

Each `/arch` option may also enable the use of other non-vector instructions that are associated with that option. An example is the use of certain BMI instructions when `/arch:AVX2` is specified.

The `__AVX__` preprocessor symbol is defined when the `/arch:AVX`, `/arch:AVX2`, `/arch:AVX512`, or `/arch:AVX10.1` compiler option is specified. The `__AVX2__` preprocessor symbol is defined when the `/arch:AVX2`, `/arch:AVX512`, or `/arch:AVX10.1` compiler option is specified. The `__AVX512F__`, `__AVX512CD__`, `__AVX512BW__`, `__AVX512DQ__`, and `__AVX512VL__` preprocessor symbols are defined when the `/arch:AVX512`, or `/arch:AVX10.1` compiler option is specified. The `__AVX10_VER__` preprocessor symbol is defined when the `/arch:AVX10.1` compiler option is specified. It indicates the AVX10 version the compiler is targeting. For more information, see Predefined macros. The `/arch:AVX2` option was introduced in Visual Studio 2013 Update 2, version 12.0.34567.1.

Limited support for `/arch:AVX512` was added in Visual Studio 2017, and expanded in Visual Studio 2019. Support for `/arch:AVX10.1` was added in Visual Studio 2022.

To set the `/arch` compiler option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify the **Enable Enhanced Instruction Set** property.

To set this compiler option programmatically

- See EnableEnhancedInstructionSet.

See also

/arch (Minimum CPU Architecture)
MSVC compiler options
MSVC compiler command-line syntax

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/arch` (ARM)

Article • 07/01/2022

Specifies the architecture for code generation on ARM. For more information on `/arch` for other target architectures, see /arch (ARM64), /arch (x64), and /arch (x86)

Syntax

> `/arch:`[`ARMv7VE`|`VFPv4`]

Arguments

`/arch:ARMv7VE`

Enables the use of ARMv7VE Virtualization Extensions instructions.

`/arch:VFPv4`

Enables the use of ARM VFPv4 instructions. If this option isn't specified, VFPv3 is the default.

Remarks

The `_M_ARM_FP` macro (for ARM only) indicates which, if any, `/arch` compiler option was used. For more information, see Predefined macros.

When you use /clr to compile, `/arch` has no effect on code generation for managed functions. `/arch` only affects code generation for native functions.

To set the `/arch:ARMv7VE` or `/arch:VFPv4` compiler option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In the **Additional options** box, add `/arch:ARMv7VE` or `/arch:VFPv4`.

To set this compiler option programmatically

- See EnableEnhancedInstructionSet.

See also

/arch (Minimum CPU architecture)
MSVC compiler options
MSVC compiler command-line syntax

`/arch` (ARM64)

Article • 06/13/2024

Specifies the Arm A-Profile architecture extension for code generation on ARM64. For more information about `/arch` for other target architectures, see /arch (x86), /arch (x64), and /arch (ARM).

Syntax

> `/arch:<armv8.0|armv8.1|armv8.2|armv8.3|armv8.4|armv8.5|armv8.6|armv8.7|armv8.8|armv8.9>` [+feature]
> `/arch:<armv9.0|armv9.1|armv9.2|armv9.3|armv9.4>` [+feature]

Arguments

`/arch:armv8.x`
Specifies the Armv8-A architecture, where `x` is a required extension value from `0` to `9`[1]. By default, the compiler uses the `/arch:armv8.0` behavior if no architecture is specified.

`/arch:armv9.x`[2]
Specifies the Armv9-A architecture, where `x` is a required extension value from `0` to `4`. By default, the compiler uses the `/arch:armv8.0` behavior if no architecture is specified.

Remarks

You can specify an ARM64 extension from Armv8.0-A through Armv8.9-A, and Armv9.0-A through Armv9.4-A. Optionally, enable one or more architecture features by appending a feature argument to the option[3]. For example, to target Armv8.0-A and enable feature `FEAT_LSE`, append feature argument `lse` so that the option becomes `/arch:armv8.0+lse`. For more information about available features and their requirements, see /feature (ARM64)[3].

> ⓘ **Note**
>
> Depending on your version of Visual Studio, the compiler may not yet generate instructions from all feature sets required by the extension level you specify. For example, `/arch:armv8.1` allows the `*Interlocked*` intrinsic functions to use the

appropriate atomic instruction introduced with the Armv8.1-A extension feature `FEAT_LSE`, but compiler support requires Visual Studio 2022 version 17.2 or later.

The `_M_ARM64` macro is defined by default when compiling for an ARM64 target. For more information, see Predefined macros\

The `__ARM_ARCH` macro is defined for `/arch:ARMv8.0` and higher. It indicates the ARM architecture extension level that the compiler is targeting. For more information, see Predefined macros.

```
C++

#if __ARM_ARCH >= 802
    // code that requires ARMv8.2...
#endif
```

`/arch` only affects code generation for native functions. When you use /clr to compile, `/arch` has no effect on code generation for managed functions.

To set the `/arch` compiler option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In the **Additional options** box, add `/arch:armv8.0` or replace `armv8.0` with a different ARM64 extension. Choose **OK** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

[1] Armv8-A architecture extension `armv8.9` is available starting in Visual Studio 2022 version 17.10.
[2] Armv9-A architecture extensions are available starting in Visual Studio 2022 version 17.10.
[3] Architecture feature enablement is available starting in Visual Studio 2022 version 17.10.

See also

/arch (Minimum CPU architecture)
Predefined macros
MSVC compiler options
MSVC compiler command-line syntax

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/await` (Enable coroutine support)

Article • 08/03/2021

Use the `/await` compiler option to enable compiler support for coroutines.

Syntax

> `/await`
>
> `/await:strict`

Remarks

The `/await` compiler option enables compiler support for C++ coroutines and the keywords `co_await`, `co_yield`, and `co_return`. This option is off by default. For information about support for coroutines in Visual Studio, see the Visual Studio Team Blog. For more information about the coroutines standard proposal, see N4628 Working Draft, Technical Specification for C++ Extensions for Coroutines.

The `/await` option is available beginning in Visual Studio 2015.

Starting in Visual Studio 2019 version 16.10, the `/await:strict` option can be used in place of `/await`. The option provides C++20-compatible coroutine support in projects that build in C++14 or C++17 mode. In `/await:strict` mode, library support is provided in <coroutine> and in the `std` namespace.

The `/await:strict` option disables language extensions present in `/await` that weren't adopted into the C++20 standard. Use of such features results in a compiler error. The option also implements coroutine behaviors such as promise parameter preview. These behaviors aren't available under `/await` because of binary compatibility issues in older versions of Visual Studio.

> **Note**
>
> Coroutine state objects obtained from `coroutine_handle<T>::address()` aren't compatible between `/await` and `/await:strict` modes. Use of `coroutine_handle<T>::from_address()` on an address obtained from a coroutine handle created by code compiled in an incompatible mode results in undefined behavior.

To set this compiler option in the Visual Studio development environment

1. Open your project's **Property Pages** dialog box.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the `/await` or `/await:strict` compiler option in the **Additional Options** box. Choose **OK** or **Apply** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/bigobj (Increase Number of Sections in .Obj file)

Article • 08/03/2021

/bigobj increases the number of sections that an object file can contain.

Syntax

> **/bigobj**

Remarks

By default, an object file can hold up to 65,279 (almost 2^16) addressable sections. This limit applies no matter which target platform is specified. **/bigobj** increases that address capacity to 4,294,967,296 (2^32).

Most modules never generate an .obj file that contains more than 65,279 sections. However, machine-generated code, or code that makes heavy use of template libraries, may require .obj files that can hold more sections. **/bigobj** is enabled by default on Universal Windows Platform (UWP) projects because the machine-generated XAML code includes a large number of headers. If you disable this option on a UWP app project, your code may generate compiler error C1128.

For information on the PE-COFF object file format, see PE Format in the Windows documentation.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the **/bigobj** compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/C (Preserve Comments During Preprocessing)

Article • 08/03/2021

Preserves comments during preprocessing.

Syntax

```
/C
```

Remarks

This compiler option requires the **/E**, **/P**, or **/EP** option.

The following code sample will display the source code comment.

C++

```
// C_compiler_option.cpp
// compile with: /E /C /c
int i;   // a variable
```

This sample will produce the following output.

```
#line 1 "C_compiler_option.cpp"
int i;   // a variable
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Preprocessor** property page.

3. Modify the **Keep Comments** property.

To set this compiler option programmatically

- See KeepComments.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
/E (Preprocess to stdout)
/P (Preprocess to a File)
/EP (Preprocess to stdout Without #line Directives)

/c (Compile Without Linking)

Article • 08/03/2021

Prevents the automatic call to LINK.

Syntax

```
/c
```

Remarks

Compiling with **/c** creates .obj files only. You must call LINK explicitly with the proper files and options to perform the linking phase of the build.

Any internal project created in the development environment uses the **/c** option by default.

To set this compiler option in the Visual Studio development environment

- This option is not available from within the development environment.

To set this compiler option programmatically

- To set this compiler option programmatically, see CompileOnly.

Example

The following command line creates the object files FIRST.obj and SECOND.obj. THIRD.obj is ignored.

```
CL /c FIRST.C SECOND.C THIRD.OBJ
```

To create an executable file, you must invoke LINK:

```
LINK firsti.obj second.obj third.obj /OUT:filename.exe
```

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/cgthreads` (Code generation threads)

Article • 08/03/2021

Sets number of cl.exe threads to use for optimization and code generation.

Syntax

> `/cgthreads1`
> `/cgthreads2`
> `/cgthreads3`
> `/cgthreads4`
> `/cgthreads5`
> `/cgthreads6`
> `/cgthreads7`
> `/cgthreads8`

Arguments

`cgthreadsN`
The maximum number of threads for cl.exe to use, where *N* is a number in the range 1 to 8.

Remarks

The `cgthreads` option specifies the maximum number of threads cl.exe uses in parallel for the optimization and code generation phases of compilation. Notice that there can be no space between `cgthreads` and the *number* argument. By default, cl.exe uses four threads, as if `/cgthreads4` were specified. If more processor cores are available, a larger *number* value can improve build times. This option is especially useful when it's combined with /GL (Whole Program Optimization).

Multiple levels of parallelism can be specified for a build. The msbuild.exe switch `/maxcpucount` specifies the number of MSBuild processes that can be run in parallel. The /MP (Build with Multiple Processes) compiler flag specifies the number of cl.exe processes that simultaneously compile the source files. The `cgthreads` option specifies the number of threads used by each cl.exe process. The processor can only run as many threads at the same time as there are processor cores. It's not useful to specify larger values for all of these options at the same time, and it can be counterproductive. For

more information about how to build projects in parallel, see Building Multiple Projects in Parallel.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `cgthreadsN`, where `N` is a value from 1 to 8, and then select **OK**.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/clr` (Common Language Runtime Compilation)

Article • 10/29/2021

Enables applications and components to use features from the common language runtime (CLR) and enables C++/CLI compilation.

Syntax

> `/clr`[`:`*options*]

Arguments

options
One or more of the following comma-separated arguments.

- none

 With no options, `/clr` creates metadata for the component. The metadata can be consumed by other CLR applications, and enables the component to consume types and data in the metadata of other CLR components. For more information, see Mixed (Native and Managed) Assemblies.

- `netcore`

 Available starting in Visual Studio 2019 version 16.4, `/clr:netcore` creates metadata and code for the component using the latest cross-platform .NET framework, also known as .NET Core. The metadata can be consumed by other .NET Core applications. And, the option enables the component to consume types and data in the metadata of other .NET Core components.

- `nostdlib`

 Instructs the compiler to ignore the default `\clr` directory. The compiler produces errors if you include multiple versions of a DLL, such as System.dll. This option lets you specify the specific framework to use during compilation.

- `pure`

`/clr:pure` **is deprecated**. The option is removed in Visual Studio 2017 and later. We recommend that you port code that must be pure MSIL to C#.

- `safe`

 `/clr:safe` **is deprecated**. The option is removed in Visual Studio 2017 and later. We recommend that you port code that must be safe MSIL to C#.

- `noAssembly`

 `/clr:noAssembly` **is deprecated**. Use /LN (Create MSIL Module) instead.

 Tells the compiler not to insert an assembly manifest into the output file. By default, the `noAssembly` option isn't in effect.

 A managed program that doesn't have assembly metadata in the manifest is known as a *module*. The `noAssembly` option can be used only to produce a module. If you compile by using /c and `/clr:noAssembly`, then specify the /NOASSEMBLY option in the linker phase to create a module.

 Before Visual Studio 2005, `/clr:noAssembly` required `/LD`. `/LD` is now implied when you specify `/clr:noAssembly`.

- `initialAppDomain`

 `initialAppDomain` **is obsolete**. Enables a C++/CLI application to run on version 1 of the CLR. An application that's compiled by using `initialAppDomain` shouldn't be used by an application that uses ASP.NET because it's not supported in version 1 of the CLR.

Remarks

Managed code is code that can be inspected and managed by the CLR. Managed code can access managed objects. For more information, see /clr Restrictions.

For information about how to develop applications that define and consume managed types in C++, see Component Extensions for Runtime Platforms.

An application compiled by using `/clr` may or may not contain managed data.

To enable debugging on a managed application, see /ASSEMBLYDEBUG (Add DebuggableAttribute).

Only CLR types are instantiated on the garbage-collected heap. For more information, see Classes and Structs. To compile a function to native code, use the `unmanaged` pragma. For more information, see managed, unmanaged.

By default, `/clr` isn't in effect. When `/clr` is in effect, `/MD` is also in effect. For more information, see /MD, /MT, /LD (Use Run-Time Library). `/MD` ensures that the dynamically linked, multithreaded versions of the runtime routines are selected from the standard header files. Multithreading is required for managed programming because the CLR garbage collector runs finalizers in an auxiliary thread.

If you compile by using `/c`, you can specify the CLR type of the resulting output file by using the /CLRIMAGETYPE linker option.

`/clr` implies `/EHa`, and no other `/EH` options are supported for `/clr`. For more information, see /EH (Exception Handling Model).

For information about how to determine the CLR image type of a file, see /CLRHEADER.

All modules passed to a given invocation of the linker must be compiled by using the same run-time library compiler option (`/MD` or `/LD`).

Use the /ASSEMBLYRESOURCE linker option to embed a resource in an assembly. /DELAYSIGN, /KEYCONTAINER, and /KEYFILE linker options also let you customize how an assembly is created.

When `/clr` is used, the `_MANAGED` symbol is defined to be 1. For more information, see Predefined macros.

The global variables in a native object file are initialized first (during `DllMain` if the executable is a DLL), and then the global variables in the managed section are initialized (before any managed code is run). #pragma init_seg only affects the order of initialization in the managed and unmanaged categories.

Metadata and Unnamed Classes

Unnamed classes appear in metadata under names such as `$UnnamedClass$<crc-of-current-file-name>$<index>$`, where `<index>` is a sequential count of the unnamed classes in the compilation. For example, the following code sample generates an unnamed class in metadata.

C++

```
// clr_unnamed_class.cpp
// compile by using: /clr /LD
```

```
class {} x;
```

Use ildasm.exe to view metadata.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Set the **Configuration** dropdown to **All configurations**, and set the **Platform** dropdown to **All Platforms**.
3. Select the **Configuration Properties** > **C/C++** > **General** page.
4. Modify the **Common Language Runtime Support** property. Choose **OK** to save your changes.

> **Note**
>
> In the Visual Studio IDE, the `/clr` compiler option can be individually set on the **Configuration Properties** > **C/C++** > **General** page of the Property Pages dialog. However, we recommend you use a CLR template to create your project. It sets all of the properties required for successful creation of a CLR component. Another way to set these properties is to use the **Common Language Runtime Support** property on the **Configuration Properties** > **Advanced** page of the Property Pages dialog. This property sets all the other CLR-related tool options at once.

To set this compiler option programmatically

- See CompileAsManaged.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/clr` Restrictions

Article • 03/02/2022

Note the following restrictions on the use of `/clr`:

- In a structured exception handler, there are restrictions on using `_alloca` when compiling with `/clr`. For more information, see _alloca.
- The use of run-time error checks isn't valid with `/clr`. For more information, see How to: Use native run-time checks.
- When `/clr` is used to compile a program that only uses standard C++ syntax, the following guidelines apply to the use of inline assembly:
 - Inline assembly code that assumes knowledge of the native stack layout, calling conventions outside of the current function, or other low-level information about the computer may fail if that knowledge is applied to the stack frame for a managed function. Functions containing inline assembly code are generated as unmanaged functions, as if they were placed in a separate module that was compiled without `/clr`.
 - Inline assembly code in functions that pass copy-constructed function parameters isn't supported.
- The vprintf Functions can't be called from a program compiled with `/clr`.
- The naked __declspec modifier is ignored under `/clr`.
- The translator function set by _set_se_translator will affect only catches in unmanaged code. For more information, see Exception handling.
- The comparison of function pointers isn't permitted under `/clr`.
- The use of functions that aren't fully prototyped isn't permitted under `/clr`.
- The following compiler options aren't supported with `/clr`:
 - `/EHsc` and `/EHs` (`/clr` implies `/EHa` (see /EH (Exception Handling Model))
 - `/fp:strict` and `/fp:except` (see /fp (Specify Floating-Point Behavior))
 - /Zd
 - /Gm

- /MT
- /RTC
- /ZI

- The combination of the `_STATIC_CPPLIB` preprocessor definition (`/D_STATIC_CPPLIB`) and the **`/clr`** compiler option isn't supported. It's because the definition would cause your application to link with the static, multithreaded C++ Standard Library, which isn't supported. For more information, see /MD, /MT, /LD (Use Run-Time Library).

- When you use **`/Zi`** with **`/clr`**, there are performance implications. For more information, see /Zi.

- Passing a wide character to a .NET Framework output routine without also specifying /Zc:wchar_t or without casting the character to **`_wchar_t`** will cause the output to appear as an `unsigned short int`. For example:

```
C++

Console::WriteLine(L' ')              // Will output 32.
Console::WriteLine((__wchar_t)L' ')   // Will output a space.
```

- /GS is ignored when compiling with **`/clr`**, unless a function is under #pragma unmanaged or if the function must be compiled as native code, in which case the compiler will generate warning C4793, which is off by default.

- See /ENTRY for function signature requirements of a managed application.

- Applications compiled with **`/openmp`** and **`/clr`** can only be run in a single appdomain process. For more information, see /openmp (Enable OpenMP 2.0 Support).

- Functions that take a variable number of arguments (varargs) will be generated as native functions. Any managed data types in the variable argument position will be marshaled to native types. Any System.String types are actually wide-character strings, but they're marshaled to single-byte character strings. So if a `printf` specifier is `%S` (`wchar_t*`), it will marshal to a `%s` string instead.

- When using the `va_arg` macro, you may get unexpected results when compiling with **`/clr:pure`**. For more information, see va_arg, va_copy, va_end, va_start. The **`/clr:pure`** and **`/clr:safe`** compiler options are deprecated in Visual Studio 2015

and unsupported in Visual Studio 2017 and later. Code that must be "pure" or "safe" should be ported to C#.

- You shouldn't call any functions that walk the stack to get parameter information (function arguments) from managed code. The P/Invoke layer causes that information to be further down the stack. For example, don't compile proxy/stub with `/clr`.

- Functions will be compiled to managed code whenever possible, but not all C++ constructs can be translated to managed code. This determination is made on a function-by-function basis. If any part of a function can't be converted to managed code, the entire function will be converted to native code instead. The following cases prevent the compiler from generating managed code.

 - Compiler-generated thunks or helper functions. Native thunks are generated for any function call through a function pointer, including virtual function calls.

 - Functions that call `setjmp` or `longjmp`.

 - Functions that use certain intrinsic routines to directly manipulate machine resources. For example, the use of `__enable` and `__disable`, `_ReturnAddress` and `_AddressOfReturnAddress`, or multimedia intrinsics will all result in native code.

 - Functions that follow the `#pragma unmanaged` directive. (The inverse, `#pragma managed`, is also supported.)

 - A function that contains references to aligned types, that is, types declared using `__declspec(align(...))`.

See also

- /clr (Common Language Runtime compilation)

/constexpr (Control constexpr evaluation)

Article • 08/03/2021

Use the **/constexpr** compiler options to control parameters for `constexpr` evaluation at compile time.

Syntax

> **/constexpr:depth***N*
> **/constexpr:backtrace***N*
> **/constexpr:steps***N*

Arguments

depth*N* Limit the depth of recursive `constexpr` function invocation to *N* levels. The default is 512.

backtrace*N* Show up to *N* `constexpr` evaluations in diagnostics. The default is 10.

steps*N* Terminate `constexpr` evaluation after *N* steps. The default is 100,000.

Remarks

The **/constexpr** compiler options control compile-time evaluation of `constexpr` expressions. Evaluation steps, recursion levels, and backtrace depth are controlled to prevent the compiler from spending too much time on `constexpr` evaluation. For more information on the `constexpr` language element, see constexpr (C++).

The **/constexpr** options are available beginning in Visual Studio 2015.

To set this compiler option in the Visual Studio development environment

1. Open your project's **Property Pages** dialog box.

2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

3. Enter any **/constexpr** compiler options in the **Additional Options** box. Choose **OK** or **Apply** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/D (Preprocessor Definitions)

Article • 08/03/2021

Defines a preprocessing symbol for a source file.

Syntax

> `/D`[]*name*[`=` | `#` [{ *string* | *number* }]]
> `/D`[]`"`*name*[`=` | `#` [{ *string* | *number* }]]`"`

Remarks

You can use this symbol together with `#if` or `#ifdef` to compile source code conditionally. The symbol definition remains in effect until it's redefined in the code, or is undefined in the code by an `#undef` directive.

`/D` has the same effect as a `#define` directive at the beginning of a source code file. The difference is that `/D` strips quotation marks on the command line, and a `#define` directive keeps them. You can have whitespace between the `/D` and the symbol. There can't be whitespace between the symbol and the equals sign, or between the equals sign and any value assigned.

By default, the value associated with a symbol is 1. For example, `/D name` is equivalent to `/D name=1`. In the example at the end of this article, the definition of `TEST` is shown to print `1`.

Compiling by using `/D name=` causes the symbol *name* to have no associated value. Although the symbol can still be used to conditionally compile code, it otherwise evaluates to nothing. In the example, if you compile by using `/DTEST=`, an error occurs. This behavior resembles the use of `#define` with or without a value.

The `/D` option doesn't support function-like macro definitions. To insert definitions that can't be defined on the command line, consider the /FI (Name forced include file) compiler option.

You can use `/D` multiple times on the command line to define more symbols. If the same symbol is defined more than once, the last definition is used.

This command defines the symbol DEBUG in TEST.c:

Windows Command Prompt

```
CL /DDEBUG TEST.C
```

This command removes all occurrences of the keyword `__far` in TEST.c:

Windows Command Prompt

```
CL /D __far= TEST.C
```

The **CL** environment variable can't be set to a string that contains the equal sign. To use `/D` together with the `CL` environment variable, you must specify the number sign (`#`) instead of the equal sign:

Windows Command Prompt

```
SET CL=/DTEST#0
```

When you define a preprocessing symbol at the command prompt, consider both compiler parsing rules and shell parsing rules. For example, to define a percent-sign preprocessing symbol (`%`) in your program, specify two percent-sign characters (`%%`) at the command prompt. If you specify only one, a parsing error is emitted.

Windows Command Prompt

```
CL /DTEST=%% TEST.C
```

To set this compiler option in the Visual Studio development environment

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Preprocessor** property page.
3. Open the drop-down menu of the **Preprocessor Definitions** property and choose **Edit**.
4. In the **Preprocessor Definitions** dialog box, add, modify, or delete one or more definitions, one per line. Choose **OK** to save your changes.

You don't need to include the '/D' option prefix on the definitions you specify here. In the property page, definitions are separated by semicolons (`;`).

To set this compiler option programmatically

- See PreprocessorDefinitions.

Example

C++

```
// cpp_D_compiler_option.cpp
// compile with: cl /EHsc /DTEST cpp_D_compiler_option.cpp
#include <stdio.h>

int main( )
{
#ifdef TEST
    printf_s("TEST defined %d\n", TEST);
#else
    printf_s("TEST not defined\n");
#endif
}
```

Output

```
TEST defined 1
```

See also

MSVC compiler options
MSVC compiler command-line syntax
/FI (Name forced include file)
/U, /u (Undefine Symbols)
#undef Directive (C/C++)
#define Directive (C/C++)

/diagnostics (Compiler diagnostic options)

Article • 08/03/2021

Use the **/diagnostics** compiler option to specify the display of error and warning location information.

Syntax

```
/diagnostics:{caret|classic|column}
```

Remarks

This option is supported in Visual Studio 2017 and later.

The **/diagnostics** compiler option controls the display of error and warning information.

The **/diagnostics:classic** option is the default, which reports only the line number where the issue was found.

The **/diagnostics:column** option also includes the column where the issue was found. This can help you identify the specific language construct or character that is causing the issue.

The **/diagnostics:caret** option includes the column where the issue was found and places a caret (^) under the location in the line of code where the issue was detected.

Note that in some cases, the compiler does not detect an issue where it occurred. For example, a missing semicolon may not be detected until other, unexpected symbols have been encountered. The column is reported and the caret is placed where the compiler detected that something was wrong, which is not always where you need to make your correction.

The **/diagnostics** option is available starting in Visual Studio 2017.

To set this compiler option in the Visual Studio development environment

1. Open your project's **Property Pages** dialog box.
2. Under **Configuration Properties**, expand the **C/C++** folder and choose the **General** property page.
3. Use the dropdown control in the **Diagnostics Format** field to select a diagnostics display option. Choose **OK** or **Apply** to save your changes.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/doc (Process Documentation Comments) (C/C++)

Article • 08/03/2021

Causes the compiler to process documentation comments in source code files and to create an .xdc file for each source code file that has documentation comments.

Syntax

> **/doc**[*name*]

Arguments

name
The name of the .xdc file that the compiler will create. Only valid when one .cpp file is passed in the compilation.

Remarks

The .xdc files are processed into an .xml file with xdcmake.exe. For more information, see XDCMake Reference.

You can add documentation comments to your source code files. For more information, see Recommended Tags for Documentation Comments.

To use the generated .xml file with IntelliSense, make the file name of the .xml file the same as the assembly that you want to support and put the .xml file is in the same directory as the assembly. When the assembly is referenced in the Visual Studio project, the .xml file is also found. For more information, see Using IntelliSense and Supplying XML Code Comments.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Output Files** property page.

3. Modify the **Generate XML Documentation Files** property.

To set this linker option programmatically

- See GenerateXMLDocumentationFiles.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/E (Preprocess to stdout)

Article • 08/03/2021

Preprocesses C and C++ source files and copies the preprocessed files to the standard output device.

Syntax

```
/E
```

Remarks

In this process, all preprocessor directives are carried out, macro expansions are performed, and comments are removed. To preserve comments in the preprocessed output, use the /C (Preserve Comments During Preprocessing) compiler option as well.

/E adds `#line` directives to the output at the beginning and end of each included file and around lines removed by preprocessor directives for conditional compilation. These directives renumber the lines of the preprocessed file. As a result, errors generated during later stages of processing refer to the line numbers of the original source file rather than lines in the preprocessed file.

The **/E** option suppresses compilation. You must resubmit the preprocessed file for compilation. **/E** also suppresses the output files from the **/FA**, **/Fa**, and **/Fm** options. For more information, see /FA, /Fa (Listing File) and /Fm (Name Mapfile).

To suppress `#line` directives, use the /EP (Preprocess to stdout Without #line Directives) option instead.

To send the preprocessed output to a file instead of to `stdout`, use the /P (Preprocess to a File) option instead.

To suppress `#line` directives and send the preprocessed output to a file, use **/P** and **/EP** together.

You cannot use precompiled headers with the **/E** option.

Note that when preprocessing to a separate file, spaces are not emitted after tokens. This can result in an illegal program or have unintended side effects. The following

program compiles successfully:

```
#define m(x) x
m(int)main( )
{
   return 0;
}
```

However, if you compile with:

```
cl -E test.cpp > test2.cpp
```

`int main` in test2.cpp will incorrectly be `intmain`.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See GeneratePreprocessedFile.

Example

The following command line preprocesses `ADD.C`, preserves comments, adds `#line` directives, and displays the result on the standard output device:

```
CL /E /C ADD.C
```

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/EH` (Exception handling model)

Article • 08/03/2021

Specifies the exception handling model support generated by the compiler. Arguments specify whether to apply `catch(...)` syntax to both structured and standard C++ exceptions, whether **extern "C"** code is assumed to throw exceptions, and whether to optimize away certain `noexcept` checks.

Syntax

> **`/EHa`**[**`-`**]
> **`/EHs`**[**`-`**]
> **`/EHc`**[**`-`**]
> **`/EHr`**[**`-`**]

Arguments

`a`

Enables standard C++ stack unwinding. Catches both structured (asynchronous) and standard C++ (synchronous) exceptions when you use `catch(...)` syntax. **`/EHa`** overrides both **`/EHs`** and **`/EHc`** arguments.

`s`

Enables standard C++ stack unwinding. Catches only standard C++ exceptions when you use `catch(...)` syntax. Unless **`/EHc`** is also specified, the compiler assumes that functions declared as **extern "C"** may throw a C++ exception.

`c`

When used with **`/EHs`**, the compiler assumes that functions declared as **extern "C"** never throw a C++ exception. It has no effect when used with **`/EHa`** (that is, **`/EHca`** is equivalent to **`/EHa`**). **`/EHc`** is ignored if **`/EHs`** or **`/EHa`** aren't specified.

`r`

Tells the compiler to always generate runtime termination checks for all **`noexcept`** functions. By default, runtime checks for **`noexcept`** may be optimized away if the compiler determines the function calls only non-throwing functions. This option gives strict C++ conformance at the cost of some extra code. **`/EHr`** is ignored if **`/EHs`** or **`/EHa`** aren't specified.

`-`

Clears the previous option argument. For example, `/EHsc-` is interpreted as `/EHs /EHc-`, and is equivalent to `/EHs`.

`/EH` arguments may be specified separately or combined, in any order. If more than one instance of the same argument is specified, the last one overrides any earlier ones. For example, `/EHr- /EHc /EHs` is the same as `/EHscr-`, and `/EHscr- /EHr` has the same effect as `/EHscr`.

Remarks

Default exception handling behavior

The compiler always generates code that supports asynchronous structured exception handling (SEH). By default (that is, if no `/EHsc`, `/EHs`, or `/EHa` option is specified), the compiler supports SEH handlers in the native C++ `catch(...)` clause. However, it also generates code that only partially supports C++ exceptions. The default exception unwinding code doesn't destroy automatic C++ objects outside of try blocks that go out of scope because of an exception. Resource leaks and undefined behavior may result when a C++ exception is thrown.

Standard C++ exception handling

Full compiler support for the Standard C++ exception handling model that safely unwinds stack objects requires `/EHsc` (recommended), `/EHs`, or `/EHa`.

If you use `/EHs` or `/EHsc`, then your `catch(...)` clauses don't catch asynchronous structured exceptions. Any access violations and managed System.Exception exceptions go uncaught. And, objects in scope when an asynchronous exception occurs aren't destroyed, even if the code handles the asynchronous exception. This behavior is an argument for leaving structured exceptions unhandled. Instead, consider these exceptions fatal.

When you use `/EHs` or `/EHsc`, the compiler assumes that exceptions can only occur at a `throw` statement or at a function call. This assumption allows the compiler to eliminate code for tracking the lifetime of many unwindable objects, which can significantly reduce code size. If you use `/EHa`, your executable image may be larger and slower, because the compiler doesn't optimize `try` blocks as aggressively. It also leaves in exception filters that automatically clean up local objects, even if the compiler doesn't see any code that can throw a C++ exception.

Structured and standard C++ exception handling

The `/EHa` compiler option enables safe stack unwinding for both asynchronous exceptions and C++ exceptions. It supports handling of both standard C++ and structured exceptions by using the native C++ `catch(...)` clause. To implement SEH without specifying `/EHa`, you may use the `__try`, `__except`, and `__finally` syntax. For more information, see Structured exception handling.

> **ⓘ Important**
>
> Specifying `/EHa` and trying to handle all exceptions by using `catch(...)` can be dangerous. In most cases, asynchronous exceptions are unrecoverable and should be considered fatal. Catching them and proceeding can cause process corruption and lead to bugs that are hard to find and fix.
>
> Even though Windows and Visual C++ support SEH, we strongly recommend that you use ISO-standard C++ exception handling (`/EHsc` or `/EHs`). It makes your code more portable and flexible. There may still be times you have to use SEH in legacy code or for particular kinds of programs. It's required in code compiled to support the common language runtime (/clr), for example. For more information, see **Structured exception handling**.
>
> We recommend that you never link object files compiled using `/EHa` to ones compiled using `/EHs` or `/EHsc` in the same executable module. If you have to handle an asynchronous exception by using `/EHa` anywhere in your module, use `/EHa` to compile all the code in the module. You can use structured exception handling syntax in the same module as code that's compiled by using `/EHs`. However, you can't mix the SEH syntax with C++ `try`, `throw`, and `catch` in the same function.

Use `/EHa` if you want to catch an exception that's raised by something other than a `throw`. This example generates and catches a structured exception:

C++

```
// compiler_options_EHA.cpp
// compile with: /EHa
#include <iostream>
#include <excpt.h>
using namespace std;

void fail()
{
```

```
    // generates SE and attempts to catch it using catch(...)
    try
    {
        int i = 0, j = 1;
        j /= i;   // This will throw a SE (divide by zero).
        printf("%d", j);
    }
    catch(...)
    {
        // catch block will only be executed under /EHa
        cout << "Caught an exception in catch(...)." << endl;
    }
}

int main()
{
    __try
    {
        fail();
    }

    // __except will only catch an exception here
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        // if the exception was not caught by the catch(...) inside fail()
        cout << "An exception was caught in __except." << endl;
    }
}
```

Exception handling under /clr

The `/clr` option implies `/EHa` (that is, `/clr /EHa` is redundant). The compiler generates an error if `/EHs` or `/EHsc` is used after `/clr`. Optimizations don't affect this behavior. When an exception is caught, the compiler invokes the class destructors for any objects that are in the same scope as the exception. If an exception isn't caught, those destructors aren't run.

For information about exception handling restrictions under `/clr`, see _set_se_translator.

Runtime exception checks

The `/EHr` option forces runtime termination checks in all functions that have a `noexcept` attribute. By default, runtime checks may be optimized away if the compiler back-end determines that a function only calls *non-throwing* functions. Non-throwing functions are any functions that have an attribute that specifies no exceptions may be thrown. They include functions marked `noexcept`, `throw()`, `__declspec(nothrow)`, and, when

`/EHc` is specified, `extern "C"` functions. Non-throwing functions also include any that the compiler has determined are non-throwing by inspection. You can explicitly set the default behavior by using `/EHr-`.

A non-throwing attribute isn't a guarantee that exceptions can't be thrown by a function. Unlike the behavior of a `noexcept` function, the MSVC compiler considers an exception thrown by a function declared using `throw()`, `__declspec(nothrow)`, or `extern "C"` as undefined behavior. Functions that use these three declaration attributes don't enforce runtime termination checks for exceptions. You can use the `/EHr` option to help you identify this undefined behavior, by forcing the compiler to generate runtime checks for unhandled exceptions that escape a `noexcept` function.

Set the option in Visual Studio or programmatically

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select **Configuration Properties** > **C/C++** > **Code Generation**.
3. Modify the **Enable C++ Exceptions** property.

 Or, set **Enable C++ Exceptions** to **No**, and then on the **Command Line** property page, in the **Additional Options** box, add the compiler option.

To set this compiler option programmatically

- See ExceptionHandling.

See also

MSVC Compiler options
MSVC Compiler command-line syntax
Errors and exception handling
Exception specifications (throw)
Structured Exception Handling (C/C++)

/EP (Preprocess to stdout Without #line Directives)

Article • 08/03/2021

Preprocesses C and C++ source files and copies the preprocessed files to the standard output device.

Syntax

```
/EP
```

Remarks

In the process, all preprocessor directives are carried out, macro expansions are performed, and comments are removed. To preserve comments in the preprocessed output, use the /C (Preserve Comments During Preprocessing) option with **/EP**.

The **/EP** option suppresses compilation. You must resubmit the preprocessed file for compilation. **/EP** also suppresses the output files from the **/FA**, **/Fa**, and **/Fm** options. For more information, see /FA, /Fa (Listing File) and /Fm (Name Mapfile).

Errors generated during later stages of processing refer to the line numbers of the preprocessed file rather than the original source file. If you want line numbers to refer to the original source file, use /E (Preprocess to stdout) instead. The **/E** option adds `#line` directives to the output for this purpose.

To send the preprocessed output, with `#line` directives, to a file, use the /P (Preprocess to a File) option instead.

To send the preprocessed output to stdout, with `#line` directives, use **/P** and **/EP** together.

You cannot use precompiled headers with the **/EP** option.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Preprocessor** property page.
3. Modify the **Generate Preprocessed File** property.

To set this compiler option programmatically

- See GeneratePreprocessedFile.

Example

The following command line preprocesses file `ADD.C`, preserves comments, and displays the result on the standard output device:

```
CL /EP /C ADD.C
```

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/errorReport (Report Internal Compiler Errors)

Article • 08/03/2021

> **Note**
>
> The **/errorReport** option is deprecated. Starting in Windows Vista, error reporting is controlled by **Windows Error Reporting (WER)** settings.

Syntax

> **/errorReport:**[**none** | **prompt** | **queue** | **send**]

Remarks

An internal compiler error (ICE) results when the compiler can't process a source code file. When an ICE occurs, the compiler doesn't produce an output file, or any useful diagnostic that you can use to fix your code.

The **/errorReport** arguments are overridden by the Windows Error Reporting service settings. The compiler automatically sends reports of internal errors to Microsoft, if reporting is enabled by Windows Error Reporting. No report is sent if disabled by Windows Error Reporting.

To set this compiler option in the Visual Studio development environment

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Advanced** property page.
3. Modify the **Error Reporting** property.

To set this compiler option programmatically

- See ErrorReporting.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/execution-charset` (Set execution character set)

Article • 02/01/2022

This option lets you specify the execution character set for your executable.

Syntax

> **`/execution-charset:`**[*`IANA_name`* | *`.CPID`*]

Arguments

`IANA_name`
The IANA-defined character set name.

`.CPID`
The code page identifier, preceded by a `.` character.

Remarks

You can use the **`/execution-charset`** option to specify an execution character set. The execution character set is the encoding used for the text of your program that is input to the compilation phase after all preprocessing steps. This character set is used for the internal representation of any string or character literals in the compiled code. Set this option to specify the extended execution character set to use when your source files include characters that are not representable in the basic execution character set. You can use either the IANA or ISO character set name, or a dot (`.`) followed by 3-5 decimal digits that specify the code page identifier of the character set to use. For a list of supported code page identifiers and character set names, see Code Page Identifiers.

By default, Visual Studio detects a byte-order mark to determine if the source file is in an encoded Unicode format, for example, UTF-16 or UTF-8. If no byte-order mark is found, it assumes that the source file is encoded in the current user code page, unless you used the **`/source-charset`** or **`/utf-8`** option to specify a character set name or code page. Visual Studio allows you to save your C++ source code in any of several character encodings. For information about source and execution character sets, see Character sets in the language documentation.

If you want to set both the source character set and the execution character set to UTF-8, you can use the `/utf-8*` compiler option as a shortcut. It's equivalent to `/source-charset:utf-8 /execution-charset:utf-8` on the command line. Any of these options also enables the `/validate-charset` option by default.

To set this compiler option in the Visual Studio development environment

1. Open the **Property Pages** dialog box for your project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In **Additional Options**, add the `/execution-charset` option, and specify your preferred encoding.
4. Choose **OK** to save your changes.

See also

MSVC compiler options
MSVC compiler command-line syntax
/source-charset (Set source character set)
/utf-8 (Set source and execution character sets to UTF-8)
/validate-charset (Validate for compatible characters)

`/experimental:log` (Structured SARIF diagnostics)

Article • 11/13/2023

Output SARIF diagnostics to the specified file. For more information, see Structured SARIF Diagnostics.

Syntax

> `/experimental:log` *filename*

Arguments

filename

Where to output SARIF diagnostics. The `.sarif` suffix is added to *filename* to produce the final filename at which to store the resulting SARIF diagnostics. The space between `/experimental:log` and *filename* is optional. Paths that include spaces must be enclosed in double quotes. *filename* may name a relative or absolute path.

Remarks

This option is available starting in Visual Studio 2022 version 17.8.

Diagnostics are also output as text to the console as usual.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the specific project **Configuration** and **Platform** for which you want to change the property. You can also choose "**All Configurations**" and "**All Platforms**".
3. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
4. Modify the **Additional Options** property, and then choose **OK**.

Example

The following command produces SARIF information for the entire compilation in the `diags.sarif` file in the current directory:

Windows Command Prompt

```
CL /experimental:logdiags main.cpp other.cpp
```

See also

Structured SARIF Diagnostics

`/experimental:module` (Enable experimental module support)

Article • 02/14/2025

Enables compiler support for Microsoft's experimental form of C++ Standard modules. This option is obsolete in Visual Studio 2019 version 16.11 and later.

Syntax

> `/experimental:module`[`-`]

Remarks

This switch was for the time before the new, standardized, way of consuming the C++ Standard Library as modules described in Import the C++ standard library using modules was available. Although you can use this switch to use the older experimental named modules, we recommend that you use the new, standardized, way of consuming the C++ Standard Library as modules described in Import the C++ standard library using modules.

This compiler became available starting in Visual Studio 2015 Update 1. Ensure that **C++ Modules for v143 build tools (x64/x86 - experimental)** in selected the VS Installer. It's available in the **Individual components** tab of the installer. Search for **experimental** to see the option. For more information, see Install C and C++ support in Visual Studio.

Expand table

Version	Status
Visual Studio 2015 Update 1	`/experimental:module` introduced.
Visual Studio 2019 version 16.10	C++20 modules support is feature complete.
Visual Studio 2019 16.11 and earlier	Enable experimental modules support using `/experimental:module` along with /std:c++latest.
Visual Studio 2019 version 16.11 and later	Modules support is enabled automatically with `/std:c++20` or later, or `/std:c++latest`. Use `/experimental:module-` to disable experimental module support.

The experimental library consists of the following named modules:

- `std.regex` provides the content of header `<regex>`
- `std.filesystem` provides the content of header `<filesystem>`
- `std.memory` provides the content of header `<memory>`
- `std.threading` provides the contents of headers `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>`, and `<thread>`
- `std.core` provides everything else in the C++ Standard Library

To consume these modules, add an import declaration to the top of the source code file. For example:

```
C++

import std.core;
import std.regex;
```

To consume the experimental Microsoft Standard Library modules, compile your program with the /EHsc and /MD options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Set the **Configuration** drop-down to **All Configurations**.
3. Select the **Configuration Properties** > **C/C++** > **Language** property page.
4. Modify the **Enable C++ Modules (experimental)** property, and then choose **OK**.

For more information about how to use and create modules, see Overview of modules in C++.

See also

/headerUnit (Use header unit IFC)
/exportHeader (Create header units)
/reference (Use named module IFC)
/translateInclude (Translate include directives into import directives)
/Zc (Conformance)

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/experimental:preprocessor` (Enable preprocessor conformance mode)

Article • 08/03/2021

This option is obsolete starting in Visual Studio 2019 version 16.5, replaced by the /Zc:preprocessor compiler option. `/experimental:preprocessor` enables an experimental, token-based preprocessor that more closely conforms to C++11 standards, including C99 preprocessor features. For more information, see MSVC new preprocessor overview.

Syntax

> `/experimental:preprocessor`[`-`]

Remarks

Use the `/experimental:preprocessor` compiler option to enable the experimental conforming preprocessor. You can use `/experimental:preprocessor-` option to explicitly specify the traditional preprocessor.

The `/experimental:preprocessor` option is available starting in Visual Studio 2017 version 15.8. Starting in Visual Studio 2019 version 16.5, the new preprocessor is complete, and available under the /Zc:preprocessor compiler option.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/experimental:preprocessor` and then choose **OK**.

See also

/Zc (Conformance)

`/external` (External headers diagnostics)

Article • 06/30/2022

The `/external` compiler options let you specify compiler diagnostic behavior for certain header files. "External" headers are the natural complement of "Just my code": Header files such as system files or third-party library files that you can't or don't intend to change. Since you aren't going to change these files, you may decide it isn't useful to see diagnostic messages from the compiler about them. The `/external` compiler options give you control over these warnings.

The `/external` compiler options are available starting in Visual Studio 2017 version 15.6. In versions of Visual Studio before Visual Studio 2019 version 16.10, the `/external` options require you also set the `/experimental:external` compiler option.

Syntax

Use external header options (Not required in 16.10 and later):

> `/experimental:external`

Specify external headers:

> `/external:anglebrackets`
> `/external:env:`*`var`*
> `/external:I` *`path`*

Specify diagnostics behavior:

> `/external:W0`
> `/external:W1`
> `/external:W2`
> `/external:W3`
> `/external:W4`
> `/external:templates-`

Arguments

`/experimental:external`

Enables the external headers options. This option isn't required in Visual Studio 2019 version 16.10 and later.

`/external:anglebrackets`

Treats all headers included by `#include <header>`, where the *`header`* file is enclosed in angle brackets (`< >`), as external headers.

`/external:I` *`path`*

Defines a root directory that contains external headers. All recursive subdirectories of *`path`* are considered external, but only the *`path`* value is added to the list of directories the compiler searches for include files. The space between **`/external:I`** and *`path`* is optional. Directories that include spaces must be enclosed in double quotes. A directory may be an absolute path or a relative path.

`/external:env:`*`var`*

Specifies the name of an environment variable *`var`* that holds a semicolon-separated list of external header directories. It's useful for build systems that rely on environment variables such as `INCLUDE`, which you use to specify the list of external include files. Or, `CAExcludePath`, for files that shouldn't be analyzed by `/analyze`. For example, you can specify `/external:env:INCLUDE` to make every directory in `INCLUDE` an external header directory at once. It's the same as using **`/external:I`** to specify the individual directories, but much less verbose. There should be no space between *`var`* and **`/external:env:`**.

`/external:Wn`

This option sets the default warning level to *`n`* (a value from 0 to 4) for external headers. For example, **`/external:W0`** effectively turns off warnings for external headers. If this option isn't specified, the compiler issues command line warning D9007 for other **`/external`** options. Those options are ignored, because they would have no effect.

The **`/external:Wn`** option has an effect similar to wrapping an included header in a `#pragma warning` directive:

```
C++

#pragma warning (push, 0)
// the global warning level is now 0 here
#include <external_header>
#pragma warning (pop)
```

`/external:templates-`

Allows warnings from external headers when they occur in a template that's instantiated

in your code.

Remarks

By default, the `/Wn` warning level you specify for your build applies to all files. The options to specify external headers only define a set of files to which you can apply a different default warning level. So, if you specify external headers, also use `/external:Wn` to specify an external warning level to change compiler behavior.

All the existing mechanisms to enable, disable, and suppress warnings still work in both external and non-external files. For example, a warning pragma can still override the default warning level you set for external headers.

Example: Set external warning level

This sample program has two source files, *`program.cpp`* and *`header_file.h`*. The *`header_file.h`* file is in an *`include_dir`* subdirectory of the directory containing the *`program.cpp`* file:

Source file `include_dir/header_file.h`:

C++

```
// External header: include_dir/header_file.h

template <typename T>
struct sample_struct
{
    static const T value = -7; // W4: warning C4245: 'initializing':
    // conversion from 'int' to 'unsigned int', signed/unsigned mismatch
};
```

Source file `program.cpp`:

C++

```
// User code: program.cpp
#include <header_file.h>

int main()
{
    return sample_struct<unsigned int>().value;
}
```

You can build the sample by using this command line:

Windows Command Prompt

```
cl /EHsc /I include_dir /W4 program.cpp
```

As expected, this sample generates a warning:

Output

```
program.cpp
include_dir\header_file.h(6): warning C4245: 'initializing': conversion from
'int' to 'const T', signed/unsigned mismatch
        with
        [
            T=unsigned int
        ]
program.cpp(6): note: see reference to class template instantiation
'sample_struct<unsigned int>' being compiled
```

To treat the header file as an external file and suppress the warning, you can use this command line instead*:

Windows Command Prompt

```
cl /EHsc /I include_dir /external:anglebrackets /external:W0 /W4 program.cpp
```

This command line suppresses the warning inside *`header_file.h`* while preserving warnings inside *`program.cpp`*.

Warnings across the internal and external boundary

Setting a low warning level for external headers can hide some actionable warnings. In particular, it can turn off warnings emitted on template instantiations in user code. These warnings might indicate a problem in your code that only happens in instantiations for particular types. (For example, if you forgot to apply a type trait removing `const` or `&`.) To avoid silencing warnings inside templates defined in external headers, you can use the **`/external:templates-`** option. The compiler considers both the effective warning level in the file that defines the template, and the warning level where template instantiation occurs. Warnings emitted inside an external template appear if the template is instantiated within non-external code. For example, this command line re-enables warnings from template sources in the sample code*:

Windows Command Prompt

```
cl /EHsc /I include_dir /external:anglebrackets /external:W0
/external:templates- /W4 program.cpp
```

The C4245 warning appears again in the output, even though the template code is inside an external header.

Enable, disable, or suppress warnings

All the existing mechanisms to enable, disable, and suppress warnings still work in external headers. When a warning appears because you use the `/external:templates-` option, you can still suppress the warning at the point of instantiation. For example, to explicitly suppress the warning in the sample that reappears because of `/external:templates-`, use a `warning` pragma directive:

C++

```
int main()
{
    #pragma warning( suppress : 4245)
    return sample_struct<unsigned int>().value;
}
```

Library writers can use the same mechanisms to enforce certain warnings, or all the warnings at certain level, if they feel those warnings should never be silenced by `/external:Wn`. For example, this version of the header file forces warning C4245 to report an error:

C++

```
// External header: include_dir/header_file.h

#pragma warning( push, 4 )
#pragma warning( error : 4245 )

template <typename T>
struct sample_struct
{
    static const T value = -7; // W4: warning C4245: 'initializing':
conversion from 'int'
                               // to 'unsigned int', signed/unsigned
mismatch
};

#pragma warning( pop )
```

With this change to the library header, the author of the library ensures that the global warning level in this header is 4, no matter what gets specified in `/external:Wn`. Now all level 4 and above warnings are reported. The library author can also force certain warnings to be errors, disabled, suppressed, or emitted only once in the header. The `/external` options don't override that deliberate choice.

`system_header` pragma

`#pragma system_header` is an intrusive marker that allows library writers to mark certain headers as external. A file containing `#pragma system_header` is considered external from the point of the pragma to the end of the file, as if it were specified as external on the command line. The compiler emits any diagnostics after the pragma at the warning level specified by `/external:Wn`. For more information, see system_header pragma.

Limitations

Some warnings emitted by the compiler's back-end code generation aren't affected by the `/external` options. These warnings usually start with C47XX, though not all C47XX warnings are back-end warnings. You can still disable these warnings individually by using `/wd47XX`. Code analysis warnings are also unaffected, since they don't have warning levels.

To set this compiler option in the Visual Studio development environment

In Visual Studio 2019 version 16.10 and later:

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **VC++ Directories** property page.
3. Set the **External Include Directories** property to specify the IDE equivalent of the `/external:I path` option for each semicolon-delimited path.
4. Select the **Configuration Properties** > **C/C++** > **External Includes** property page.
5. Set properties:
 - Set **Treat Files Included with Angle Brackets as External** to **Yes** to set the `/external:anglebrackets` option.

- **External Header Warning Level** allows you to set the `/external:Wn` option. If this value is set to **Inherit Project Warning Level** or the default, other `/external` options are ignored.
- Set **Template Diagnostics in External Headers** to **Yes** to set the `/external:templates-` option.

6. Choose **OK** or **Apply** to save your changes.

In versions of Visual Studio before Visual Studio 2019 version 16.10:

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the `/experimental:external` option and other `/external` compiler options in the **Additional Options** box.
4. Choose **OK** or **Apply** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

* Add the `/experimental:external` option to enable the external headers options in versions of Visual Studio before Visual Studio 2019 version 16.10.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/F` (Set Stack Size)

Article • 08/03/2021

Sets the program stack size in bytes.

Syntax

> `/F` *`number`*

Arguments

`number`
The stack size in bytes.

Remarks

Without this option, the stack size defaults to 1 MB. The *`number`* argument can be in decimal or C-language notation. The argument can range from 1 to the maximum stack size accepted by the linker. The linker rounds up the specified value to the nearest multiple of 4 bytes. The space between `/F` and *`number`* is optional.

You may need to increase the stack size if your program gets stack-overflow messages at runtime.

You can also set the stack size by:

- Using the `/STACK` linker option. For more information, see /STACK (Stack allocations).
- Using EDITBIN on the EXE file. For more information, see EDITBIN reference.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax

Output-File (/F) Options

Article • 08/03/2021

The output-file options create or rename output files. They affect all C or C++ source files specified in the CL environment variable, on the command line, or in any command file.

- /FA, /Fa (Listing File)
- Specifying the Pathname
- /Fd (Name PDB File)
- /Fe (Name EXE File)
- /FI (Name Forced Include File)
- /Fm (Name Mapfile)
- /Fo (Name Object File)
- /Fp (Name .pch File)
- /FR, /Fr (Create .sbr File)
- /FU (Name Forced #using File)
- /Fx (Merge Injected Code)

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/FA`, `/Fa` (Listing file)

Article • 08/03/2021

Creates a listing file containing assembler code.

Syntax

> **`/FA`**[**`c`**][**`s`**][**`u`**]
> **`/Fa`***pathname*

Remarks

The **`/FA`** compiler option generates an assembler listing file for each translation unit in the compilation, which generally corresponds to a C or C++ source file. By default, only assembler is included in the listing file, which is encoded as ANSI. The optional **`c`**, **`s`**, and **`u`** arguments to **`/FA`** control whether machine code or source code are output together with the assembler listing, and whether the listing is encoded as UTF-8.

By default, each listing file gets the same base name as the source file, and has an *`.asm`* extension. When machine code is included by using the **`c`** option, the listing file has a *`.cod`* extension. You can change the name and extension of the listing file and the directory where it's created by using the **`/Fa`** option.

`/FA` arguments

none
Only assembler language is included in the listing.

`c`
Optional. Includes machine code in the listing.

`s`
Optional. Includes source code in the listing.

`u`
Optional. Encodes the listing file in UTF-8 format, and includes a byte order marker. By default, the file is encoded as ANSI. Use **`u`** to create a listing file that displays correctly on any system, or if you're using Unicode source code files as input to the compiler.

If both `s` and `u` are specified, and if a source code file uses a Unicode encoding other than UTF-8, then the code lines in the `.asm` file may not display correctly.

`/Fa` argument

none
One *source.asm* file is created for each source code file in the compilation.

filename
The compiler places a listing file named *filename*.asm in the current directory. This argument form is only valid when compiling a single source code file.

filename.extension
The compiler places a listing file named *filename.extension* in the current directory. This argument form is only valid when compiling a single source code file.

directory\
The compiler creates one *source_file.asm* file for each source code file in the compilation. It's placed in the specified *directory*. The trailing backslash is required. Only paths on the current disk are allowed.

*directory**filename*
A listing file named *filename.asm* is placed in the specified *directory*. This argument form is only valid when compiling a single source code file.

*directory**filename.extension*
A listing file named *filename.extension* is placed in the specified *directory*. This argument form is only valid when compiling a single source code file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Output Files** property page.

3. Modify the **Assembler Output** property to set the **/FAc** and **/FAs** options for assembler, machine, and source code. Modify the **Use Unicode For Assembler Listing** property to set the `/FAu` option for ANSI or UTF-8 output. Modify the **ASM List Location** to set the `/Fa` option for listing file name and location.

Setting both **Assembler Output** and **Use Unicode For Assembler Listing** properties can cause Command-Line Warning D9025. To combine these options in the IDE, use the **Additional Options** field in the **Command Line** property page instead.

To set this compiler option programmatically

- See AssemblerListingLocation or AssemblerOutput. To specify **/FAu**, see AdditionalOptions.

Example

The following command line produces a combined source and machine-code listing called `HELLO.cod`:

Windows Command Prompt

```
CL /FAcs HELLO.CPP
```

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Specifying the Pathname

Specifying the Pathname

Article • 08/03/2021

Each output-file option accepts a *pathname* argument that can specify a location and a name for the output file. The argument can include a drive name, directory, and file name. No space is allowed between the option and the argument.

If *pathname* includes a file name without an extension, the compiler gives the output a default extension. If *pathname* includes a directory but no file name, the compiler creates a file with a default name in the specified directory. The default name is based on the base name of the source file and a default extension based on the type of the output file. If you leave off *pathname* entirely, the compiler creates a file with a default name in a default directory.

Alternatively, the *pathname* argument can be a device name (AUX, CON, PRN, or NUL) rather than a file name. Do not use a space between the option and the device name or a colon as part of the device name.

Device name	Represents
AUX	Auxiliary device
CON	Console
PRN	Printer
NUL	Null device (no file created)

Example

The following command line sends a mapfile to the printer:

```
CL /FmPRN HELLO.CPP
```

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/FD (IDE Minimal Rebuild)

Article • 08/03/2021

/FD is only exposed to users in the Command Line property page of a C++ project's **Property Pages** dialog box. It's available if and only if the deprecated and off-by-default /Gm (Enable Minimal Rebuild) option isn't selected. **/FD** has no effect other than from the development environment. **/FD** isn't exposed in the output of `cl /?`.

If you don't enable the deprecated **/Gm** option in the development environment, **/FD** is used. **/FD** ensures the .idb file has sufficient dependency information. **/FD** is only used by the development environment, and it shouldn't be used from the command line or a build script.

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Fd (Program Database File Name)

Article • 08/03/2021

Specifies a file name for the program database (PDB) file created by /Z7, /Zi, /ZI (Debug Information Format).

Syntax

```
/Fdpathname
```

Remarks

Without **/Fd**, the PDB file name defaults to VC*x*0.pdb, where *x* is the major version of Visual C++ in use.

If you specify a path name that does not include a file name (the path ends in backslash), the compiler creates a .pdb file named VC*x*0.pdb in the specified directory.

If you specify a file name that does not include an extension, the compiler uses .pdb as the extension.

This option also names the state (.idb) file used for minimal rebuild and incremental compilation.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Output Files** property page.
3. Modify the **Program Database File Name** property.

To set this compiler option programmatically

- See ProgramDataBaseFileName.

Example

This command line creates a .pdb file named PROG.pdb and an .idb file named PROG.idb:

```
CL /DDEBUG /Zi /FdPROG.PDB PROG.CPP
```

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Specifying the Pathname

/Fe (Name EXE File)

Article • 11/11/2021

Specifies a name and a directory for the .exe file or DLL created by the compiler.

Syntax

> **/Fe**[*pathname*]
> **/Fe:** *pathname*

Arguments

pathname
The relative or absolute path and base file name, or relative or absolute path to a directory, or base file name to use for the generated executable.

Remarks

The **/Fe** option allows you to specify the output directory, output executable name, or both, for the generated executable file. If *pathname* ends in a path separator (`\`), it is assumed to specify only the output directory. Otherwise, the last component of *pathname* is used as the output file base name, and the rest of *pathname* specifies the output directory. If *pathname* does not have any path separators, it's assumed to specify the output file name in the current directory. The *pathname* must be enclosed in double quotes (") if it contains any characters that can't be in a short path, such as spaces, extended characters, or path components more than eight characters long.

When the **/Fe** option is not specified, or when a file base name is not specified in *pathname*, the compiler gives the output file a default name using the base name of the first source or object file specified on the command line and the extension .exe or .dll.

If you specify the /c (Compile Without Linking) option, **/Fe** has no effect.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Linker** > **General** property page.

3. Modify the **Output File** property. Choose **OK** to save your changes.

To set this compiler option programmatically

- See OutputFile.

Examples

The following command line compiles and links all C source files in the current directory. The resulting executable file is named PROCESS.exe and is created in the directory "C:\Users\User Name\repos\My Project\bin".

```
CL /Fe"C:\Users\User Name\repos\My Project\bin\PROCESS" *.C
```

The following command line creates an executable file in `C:\BIN` with the same base name as the first source file in the current directory:

```
CL /FeC:\BIN\ *.C
```

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Specifying the Pathname

`/Fi` (Preprocess output file name)

Article • 08/03/2021

Specifies the name of the output file to which the /P (Preprocess to a File) compiler option writes preprocessed output.

Syntax

> **`/Fi`***`pathname`*

Parameters

`pathname`
The relative or absolute path and filename of the output file produced by the **`/P`** compiler option. Or, the directory path for the **`.i`** output files when more than one input file is specified. Don't put a space between the **`/Fi`** option and ***`pathname`***.

Remarks

Use the **`/Fi`** compiler option in combination with the **`/P`** compiler option. If **`/P`** isn't specified, **`/Fi`** causes command line warning D9007.

If you specify only a directory path (a path that ends in a backslash **`\`**) for the ***`pathname`*** parameter, the base name of the source file is used as the base name of the preprocessed output file. The ***`pathname`*** parameter doesn't require a particular file name extension. However, an extension of ".i" is used if you don't specify a file name extension.

Example

The following command line preprocesses ***`PROGRAM.cpp`***, preserves comments, adds #line directives, and writes the result to the ***`MYPROCESS.i`*** file:

Windows Command Prompt

```
CL /P /FiMYPROCESS.I PROGRAM.CPP
```

This command line preprocesses *main.cpp* and *helper.cpp* into *main.i* and *helper.i* in a subdirectory named *preprocessed*:

Windows Command Prompt

```
CL /P /Fi".\\preprocessed\\" main.cpp helper.cpp
```

To set this compiler option in the Visual Studio development environment

1. Open the source file or the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Preprocessor** property page.
3. Set the **Preprocess to a File** property to **Yes**.
4. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
5. Enter the **/Fi** compiler option and *pathname* in the **Additional Options** box. Only specify a directory path, not a filename, when setting this property for a project.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
/P (Preprocess to a file)
Specifying the pathname

/FI (Name Forced Include File)

Article • 08/03/2021

Causes the preprocessor to process the specified header file.

Syntax

```
/FI[ ]pathname
```

Remarks

This option has the same effect as specifying the file with double quotation marks in an `#include` directive on the first line of every source file specified on the command line, in the CL environment variable, or in a command file. If you use multiple **/FI** options, files are included in the order they are processed by CL.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Advanced** property page.
3. Modify the **Forced Include File** property.

To set this compiler option programmatically

- See ForcedIncludeFiles.

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Specifying the Pathname

/Fm (Name Mapfile)

Article • 08/03/2021

Tells the linker to produce a mapfile containing a list of segments in the order in which they appear in the corresponding .exe file or DLL.

Syntax

```
/Fmpathname
```

Remarks

By default, the map file is given the base name of the corresponding C or C++ source file with a `.MAP` extension.

Specifying **/Fm** has the same effect as if you had specified the /MAP (Generate Mapfile) linker option.

If you specify /c (Compile Without Linking) to suppress linking, **/Fm** has no effect.

Global symbols in a map file usually have one or more leading underscores. It's because the compiler adds a leading underscore to variable names. Many global symbols that appear in the map file are used internally by the compiler and the standard libraries.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Specifying the Pathname

`/Fo` (Object File Name)

Article • 01/30/2024

Specifies an object (*`.obj`*) file name or directory to be used instead of the default.

Syntax

```
/Fo"pathname"
/Fo:[ ]"pathname"
```

Remarks

You can use the **`/Fo`** compiler option to set an output directory for all the object files generated by the CL compiler command. Or, you can use it to rename a single object file. Don't put a space between the **`/Fo`** option and the *`pathname`* argument.

By default, the object files generated by the compiler are placed in the current directory. They're given the base name of the source file and a *`.obj`* extension.

To use the **`/Fo`** option to rename an object file, specify the output filename as the *`pathname`* argument. When you rename an object file, you can use any name and extension you want, but the recommended convention is to use an *`.obj`* extension. The compiler generates command line error D8036 if you specify a filename to **`/Fo`** when you've specified more than one source file to compile.

To use the **`/Fo`** option to set an output directory for all object files created by the CL command, specify the directory as the *`pathname`* argument. A directory is indicated by a trailing slash or backslash in the *`pathname`* argument. Use an escaped backslash (a double backslash), if you're using a quoted path. The directory path can be absolute, or relative to the source directory. The specified directory must exist, or the compiler reports error D8003. The directory isn't created automatically.

Example

This command line demonstrates the format that allows for an optional space between the `/Fo` option and the *`pathname`* argument. It creates an object file named *`test.obj`* in the current directory.

```
Windows Command Prompt
```

```
CL /Fo: "test" /EHsc /c sample1.cpp
```

The following command line creates object files named `sample1.obj` and `sample2.obj` in an existing directory, `D:\intermediate\`. It uses escaped backslash characters as path segment separators in a quoted path:

Windows Command Prompt

```
CL /Fo"D:\\intermediate\\" /EHsc /c sample1.cpp sample2.cpp
```

This command line creates object files named `sample1.obj` and `sample2.obj` in an existing directory, `output\`, relative to the source directory.

Windows Command Prompt

```
CL /Fooutput\ /EHsc /c sample1.cpp sample2.cpp
```

Set the option in Visual Studio or programmatically

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Output Files** property page.
3. Modify the **Object File Name** property to set the output directory. In the IDE, the object files must have an extension of `.obj`.

To set this compiler option programmatically

- See ObjectFile.

See also

Output-file (/F) options
MSVC compiler options

/Fp (Name .pch file)

Article • 12/06/2021

Provides a path name for a precompiled header instead of using the default path name.

Syntax

> **/Fp***pathname*

Remarks

Use the **/Fp** option with /Yc (Create Precompiled Header File) or /Yu (Use Precompiled Header File) to specify the path and file name for the precompiled header (PCH) file. By default, the **/Yc** option creates a PCH file name by using the base name of the source file and a *pch* extension.

If you don't specify an extension as part of the *pathname*, an extension of *pch* is assumed. When you specify a directory name by use of a slash (**/**) at the end of *pathname*, the default file name is vc*version*0.pch, where *version* is the major version of the Visual Studio toolset. This directory must exist, or error C1083 is generated.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Precompiled Headers** property page.
3. Modify the **Precompiled Header Output File** property.

To set this compiler option programmatically

- See AdditionalOptions.

Examples

To create a separate named version of the precompiled header file for the debug build of your program, you can specify a command such as:

```
CMD

CL /DDEBUG /Zi /Yc /FpDPROG.PCH PROG.CPP
```

The following command specifies the use of a precompiled header file named MYPCH.pch. The compiler precompiles the source code in PROG.cpp through the end of MYAPP.h, and puts the precompiled code in MYPCH.pch. It then uses the content of MYPCH.pch and compiles the rest of PROG.cpp to create an .obj file. The output of this example is a file named PROG.exe.

```
CMD

CL /YuMYAPP.H /FpMYPCH.PCH PROG.CPP
```

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Specifying the Pathname

`/FR`, `/Fr` (Name SBR file)

Article • 08/31/2022

Creates `.sbr` (source browser) files, used by Code maps, BSCMAKE, and some third-party code browsing tools.

Syntax

> `/FR`[*`pathname`*[`\`*`filename`*]]
> `/Fr`[*`pathname`*[`\`*`filename`*]]

Arguments

`pathname`
The optional destination directory for the generated `.sbr` files. If this value isn't specified, the files are created in the default output directory. For more information, see Specifying the pathname.

`filename`
An optional filename for the generated `.sbr` file. If this value isn't specified, the compiler uses the base name of the source file with a `.sbr` extension. For more information, see Specifying the pathname.

Remarks

`/FR` creates an `.sbr` file with complete symbolic information.

`/Fr` creates an `.sbr` file without information on local variables. `/Fr` is deprecated; use `/FR` instead. For more information, see the Deprecated and removed compiler options section in Compiler options listed by category.

The Visual Studio Code Maps feature requires the `.sbr` files generated by `/FR`.

The Microsoft Browse Information File Maintenance Utility (BSCMAKE) uses `.sbr` files to create a `.bsc` file, used to display browse information in some third-party tools. For more information, see BSCMAKE reference.

> ⓘ **Note**

Although BSCMAKE is still installed with Visual Studio, it's no longer used by the IDE. Since Visual Studio 2008, browse and symbol information is stored automatically in a SQL Server SDF file in the solution folder. If you use BSCMAKE, don't change the `.sbr` extension. BSCMAKE requires the intermediate files to have that extension.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Browse Information** property page.
3. Modify the **Browse Information File** or **Enable Browse Information** property.

To set this compiler option programmatically

- See BrowseInformation and BrowseInformationFile.

See also

Output-File (/F) options
MSVC compiler options
MSVC compiler command-line syntax
Specifying the pathname

/FU (Name Forced #using File)

Article • 08/03/2021

A compiler option that you can use as an alternative to passing a file name to #using Directive in source code.

Syntax

> **/FU** *file*

Arguments

file
Specifies the metadata file to reference in this compilation.

Remarks

The /FU switch takes just one file name. To specify multiple files, use /FU with each one.

If you are using C++/CLI and are referencing metadata to use the Friend Assemblies feature, you can't use **/FU**. You must reference the metadata in code by using `#using`—together with the `[as friend]` attribute. Friend assemblies are not supported in Visual C++ component extensions C++/CX.

For information about how to create an assembly or module for the common language runtime (CLR), see /clr (Common Language Runtime Compilation). For information about how to build in C++/CX, see Building apps and libraries.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Advanced** property page.
3. Modify the **Force #using** property.

To set this compiler option programmatically

- See ForcedUsingFiles.

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Fx (Merge Injected Code)

Article • 08/03/2021

Produces a copy of each source file with injected code merged into the source.

Syntax

```
/Fx
```

Remarks

To distinguish a merged source file from an original source file, **/Fx** adds an .mrg extension between the file name and file extension. For example, a file named MyCode.cpp containing attributed code and built with **/Fx** creates a file named MyCode.mrg.cpp containing the following code:

```
//+++ Start Injected Code
[no_injected_text(true)];      // Suppress injected text, it has
                               // already been injected
#pragma warning(disable: 4543) // Suppress warnings about skipping
                               // injected text
#pragma warning(disable: 4199) // Suppress warnings from attribute
                               // providers
//--- End Injected Code
```

In an .mrg file, code that was injected because of an attribute will be delimited as follows:

```
//+++ Start Injected Code
...
//--- End Injected Code
```

The no_injected_text attribute is embedded in an .mrg file, which allows for the compilation of the .mrg file without text being reinjected.

You should be aware that the .mrg source file is intended to be a representation of the source code injected by the compiler. The .mrg file may not compile or run exactly as the original source file.

Macros are not expanded in the .mrg file.

If your program includes a header file that uses injected code, **/Fx** generates an .mrg.h file for that header. **/Fx** does not merge include files that do not use injected code.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Output Files** property page.
3. Modify the **Expand Attributed Source** property.

To set this compiler option programmatically

- See ExpandAttributedSource.

See also

Output-File (/F) Options
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/favor (Optimize for Architecture Specifics)

Article • 08/03/2021

/favor: `option` produces the code that is optimized for a specific architecture or for the specifics of micro-architectures in the AMD and the Intel architectures.

Syntax

> **/favor:**{**blend** | **ATOM** | **AMD64** | **INTEL64**}

Remarks

/favor:blend
(x86 and x64) produces the code that is optimized for the specifics of micro-architectures in the AMD and the Intel architectures. While **/favor:blend** may not give the best performance possible on a specific processor, it is designed to give the best performance across a broad range of x86 and x64 processors. By default, **/favor:blend** is in effect.

/favor:ATOM
(x86 and x64) produces the code that is optimized for the specifics of the Intel Atom processor and Intel Centrino Atom Processor Technology. Code that is generated by using **/favor:ATOM** may also produce Intel SSSE3, SSE3, SSE2, and SSE instructions for Intel processors.

/favor:AMD64
(x64 only) optimizes the generated code for the AMD Opteron, and Athlon processors that support 64-bit extensions. The optimized code can run on all x64 compatible platforms. Code that is generated by using **/favor:AMD64** might cause worse performance on Intel processors that support Intel64.

/favor:INTEL64
(x64 only) optimizes the generated code for Intel processors that support Intel64, which typically yields better performance for that platform. The resulting code can run on any x64 platform. Code that is generated with **/favor:INTEL64** might cause worse performance on AMD Opteron, and Athlon processors that support 64-bit extensions.

> ⓘ **Note**

Intel64 architecture was previously known as Extended Memory 64 Technology, and the corresponding compiler option was **/favor:EM64T**.

For information about how to program for the x64 architecture, see x64 Software Conventions.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/FC` (Full path of source code file in diagnostics)

Article • 10/29/2021

Causes the compiler to display the full path of source code files passed to the compiler in diagnostics.

Syntax

> `/FC`

Remarks

Consider the following code sample, where the source file is located in *`C:\test\compiler_option_FC.cpp`*:

```
C++

// compiler_option_FC.cpp

int main( ) {
   int i   // C2143
}
```

Without `/FC`, the compiler output looks similar to this diagnostic text:

- compiler_option_FC.cpp(5): error C2143: syntax error: missing ';' before '}'

With `/FC`, the compiler output looks similar to this diagnostic text:

- C:\test\compiler_option_FC.cpp(5): error C2143: syntax error: missing ';' before '}'

`/FC` is also needed if you want to see the full path of a file name when using the `__FILE__` macro. For more information about `__FILE__`, see Predefined macros.

The `/FC` option is implied by `/ZI`. For more information about `/ZI`, see /Z7, /Zi, /ZI (Debug information format).

In Visual Studio 2017 and earlier versions, `/FC` outputs full paths in lower case. Starting in Visual Studio 2019, `/FC` uses the same casing as the file system for full paths.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Advanced** property page.
3. Modify the **Use Full Paths** property.

To set this compiler option programmatically

- See UseFullPaths.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/feature` (ARM64)

Article • 08/14/2024

Enable one or more Arm A-Profile architecture features for an ARM64 extension as specified by `/arch` (ARM64). For more information about `/arch` (ARM64), see /arch (ARM64).

Syntax

> `/feature:<arg1>`[`+arg2`]

Arguments

To enable one or more features the targeted ARM64 extension supports, specify one or more of the following feature arguments:

Expand table

Feature argument	Feature identifier	Optional from	Enabled by default	Description	Supported in version
`lse`	`FEAT_LSE`	Armv8.0	Armv8.1	Large System Extensions.	Visual Studio 2022 17.10
`rcpc`	`FEAT_LRCPC`	Armv8.2	Armv8.3	Load-Acquire RCpc instructions.	Visual Studio 2022 17.10
`rcpc2`	`FEAT_LRCPC2`	Armv8.2	Armv8.4	Load-Acquire RCpc instructions v2.	Visual Studio 2022 17.11

Remarks

Example usage: to enable `FEAT_LSE`, specify `/feature:lse`.

If there are conflicting feature arguments specified by `/feature`, the right-most feature is enabled. Enabling a feature the targeted ARM64 extension doesn't support may cause unexpected behavior, especially if a CPU doesn't implement the feature.

Use either `/feature` or only `/arch` (ARM64) to specify features. For example, to enable `FEAT_LSE` when targeting Armv8.0-A, use both `/feature:lse` and `/arch:armv8.0`, or

specify `/arch:armv8.0+lse`. `/feature` is a way to specify features without specifying them in `/arch` (ARM64).

To set the `/feature` compiler option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In the **Additional options** box, add `/feature:lse` or replace `lse` with the feature to enable. Choose **OK** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/arch (Minimum CPU architecture)
MSVC compiler options
MSVC compiler command-line syntax

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/fp` (Specify floating-point behavior)

Article • 11/22/2021

Specifies how the compiler treats floating-point expressions, optimizations, and exceptions. The `/fp` options specify whether the generated code allows floating-point environment changes to the rounding mode, exception masks, and subnormal behavior, and whether floating-point status checks return current, accurate results. It controls whether the compiler generates code that maintains source operation and expression order, and conforms to the standard for NaN propagation. Or, if it instead generates more efficient code that may reorder or combine operations and use simplifying algebraic transformations that aren't allowed by the IEEE-754 standard.

Syntax

`/fp:contract`

`/fp:except`[`-`]

`/fp:fast`

`/fp:precise`

`/fp:strict`

Arguments

`/fp:contract`

The `/fp:contract` option allows the compiler to generate floating-point contractions when you specify the `/fp:precise` and `/fp:except` options. A *contraction* is a machine instruction that combines floating-point operations, such as Fused-Multiply-Add (FMA). FMA, defined as a basic operation by IEEE-754, doesn't round the intermediate product before the addition, so the result can differ from separate multiplication and addition operations. Since it's implemented as a single instruction, it can be faster than separate instructions. The speed comes at a cost of bitwise exact results, and an inability to examine the intermediate value.

By default, the `/fp:fast` option enables `/fp:contract`. The `/fp:contract` option isn't compatible with `/fp:strict`.

The `/fp:contract` option is new in Visual Studio 2022.

`/fp:precise`

By default, the compiler uses `/fp:precise` behavior.

Under `/fp:precise`, the compiler preserves the source expression ordering and rounding properties of floating-point code when it generates and optimizes object code for the target machine. The compiler rounds to source code precision at four specific points during expression evaluation: at assignments, typecasts, when floating-point arguments get passed to a function call, and when a function call returns a floating-point value. Intermediate computations may be performed at machine precision. Typecasts can be used to explicitly round intermediate computations.

The compiler doesn't perform algebraic transformations on floating-point expressions, such as reassociation or distribution, unless it can guarantee the transformation produces a bitwise identical result. Expressions that involve special values (NaN, +infinity, -infinity, -0.0) are processed according to IEEE-754 specifications. For example, `x != x` evaluates to `true` if `x` is NaN. Floating-point contractions aren't generated by default under `/fp:precise`. This behavior is new in Visual Studio 2022. Previous compiler versions could generate contractions by default under `/fp:precise`.

The compiler generates code intended to run in the default floating-point environment. It also assumes the floating-point environment isn't accessed or modified at runtime. That is, it assumes the code: leaves floating-point exceptions masked, doesn't read or write floating-point status registers, and doesn't change rounding modes.

If your floating-point code doesn't depend on the order of operations and expressions in your floating-point statements (for example, if you don't care whether `a * b + a * c` is computed as `(b + c) * a` or `2 * a` as `a + a`), consider the /fp:fast option, which can produce faster, more efficient code. If your code both depends on the order of operations and expressions, and accesses or alters the floating-point environment (for example, to change rounding modes or to trap floating-point exceptions), use /fp:strict.

`/fp:strict`

`/fp:strict` has behavior similar to `/fp:precise`, that is, the compiler preserves the source ordering and rounding properties of floating-point code when it generates and optimizes object code for the target machine, and observes the standard when handling special values. The program may also safely access or modify the floating-point environment at runtime.

Under `/fp:strict`, the compiler generates code that allows the program to safely unmask floating-point exceptions, read or write floating-point status registers, or

change rounding modes. It rounds to source code precision at four specific points during expression evaluation: at assignments, typecasts, when floating-point arguments get passed to a function call, and when a function call returns a floating-point value. Intermediate computations may be performed at machine precision. Typecasts can be used to explicitly round intermediate computations. The compiler doesn't make any algebraic transformations on floating-point expressions, such as reassociation or distribution, unless it can guarantee the transformation produces a bitwise identical result. Expressions that involve special values (NaN, +infinity, -infinity, -0.0) are processed according to IEEE-754 specifications. For example, `x != x` evaluates to **`true`** if `x` is NaN. Floating-point contractions aren't generated under **`/fp:strict`**.

`/fp:strict` is computationally more expensive than **`/fp:precise`** because the compiler must insert extra instructions to trap exceptions and allow programs to access or modify the floating-point environment at runtime. If your code doesn't use this capability, but requires source code ordering and rounding, or relies on special values, use **`/fp:precise`**. Otherwise, consider using **`/fp:fast`**, which can produce faster and smaller code.

`/fp:fast`

The **`/fp:fast`** option allows the compiler to reorder, combine, or simplify floating-point operations to optimize floating-point code for speed and space. The compiler may omit rounding at assignment statements, typecasts, or function calls. It may reorder operations or make algebraic transforms, for example, by use of associative and distributive laws. It may reorder code even if such transformations result in observably different rounding behavior. Because of this enhanced optimization, the result of some floating-point computations may differ from the ones produced by other **`/fp`** options. Special values (NaN, +infinity, -infinity, -0.0) may not be propagated or behave strictly according to the IEEE-754 standard. Floating-point contractions may be generated under **`/fp:fast`**. The compiler is still bound by the underlying architecture under **`/fp:fast`**, and more optimizations may be available through use of the /arch option.

Under **`/fp:fast`**, the compiler generates code intended to run in the default floating-point environment and assumes the floating-point environment isn't accessed or modified at runtime. That is, it assumes the code: leaves floating-point exceptions masked, doesn't read or write floating-point status registers, and doesn't change rounding modes.

`/fp:fast` is intended for programs that don't require strict source code ordering and rounding of floating-point expressions, and don't rely on the standard rules for handling special values such as `NaN`. If your floating-point code requires preservation of source

code ordering and rounding, or relies on standard behavior of special values, use /fp:precise. If your code accesses or modifies the floating-point environment to change rounding modes, unmask floating-point exceptions, or check floating-point status, use /fp:strict.

`/fp:except`

The `/fp:except` option generates code to ensures that any unmasked floating-point exceptions are raised at the exact point at which they occur, and that no other floating-point exceptions are raised. By default, the `/fp:strict` option enables `/fp:except`, and `/fp:precise` doesn't. The `/fp:except` option isn't compatible with `/fp:fast`. The option can be explicitly disabled by use of `/fp:except-`.

By itself, `/fp:except` doesn't enable any floating-point exceptions. However, it's required for programs to enable floating-point exceptions. For more information on how to enable floating-point exceptions, see _controlfp.

Remarks

Multiple `/fp` options can be specified in the same compiler command line. Only one of `/fp:strict`, `/fp:fast`, and `/fp:precise` options can be in effect at a time. If you specify more than one of these options on the command line, the later option takes precedence and the compiler generates a warning. The `/fp:strict` and `/fp:except` options aren't compatible with `/clr`.

The /Za (ANSI compatibility) option isn't compatible with `/fp`.

Using compiler directives to control floating-point behavior

The compiler provides three pragma directives to override the floating-point behavior specified on the command line: float_control, fenv_access, and fp_contract. You can use these directives to control floating-point behavior at function-level, not within a function. These directives don't correspond directly to the `/fp` options. This table shows how the `/fp` options and pragma directives map to each other. For more information, see the documentation for the individual options and pragma directives.

Option	`float_control(precise, *)`	`float_control(except, *)`	`fenv_access(*)`	`fp_contract(*)`

Option	`float_control(precise, *)`	`float_control(except, *)`	`fenv_access(*)`	`fp_contract(*)`
`/fp:fast`	`off`	`off`	`off`	`on`
`/fp:precise`	`on`	`off`	`off`	`off`*
`/fp:strict`	`on`	`on`	`on`	`off`

* In versions of Visual Studio before Visual Studio 2022, the `/fp:precise` behavior defaulted to `fp_contract(on)`.

The default floating point environment

When a process is initialized, the *default floating point environment* is set. This environment masks all floating point exceptions, sets the rounding mode to round to nearest (`FE_TONEAREST`), preserves subnormal (denormal) values, uses the default precision of significand (mantissa) for `float`, `double`, and `long double` values, and where supported, sets the infinity control to the default affine mode.

Floating-point environment access and modification

The Microsoft Visual C++ runtime provides several functions to access and modify the floating-point environment. These include _controlfp, _clearfp, and _statusfp and their variants. To ensure correct program behavior when your code accesses or modifies the floating-point environment, `fenv_access` must be enabled, either by the `/fp:strict` option or by use of the `fenv_access` pragma, for these functions to have any effect. When `fenv_access` isn't enabled, access or modification of the floating-point environment may result in unexpected program behavior:

- Code may not honor requested changes to the floating-point environment,
- The floating-point status registers may not report expected or current results,
- Unexpected floating-point exceptions may occur or expected floating-point exceptions may not occur.

When your code accesses or modifies the floating-point environment, you must be careful when you combine code where `fenv_access` is enabled with code that doesn't have `fenv_access` enabled. In code where `fenv_access` isn't enabled, the compiler assumes the platform default floating-point environment is in effect. It also assumes the floating-point status isn't accessed or modified. We recommend you save and restore

the local floating-point environment to its default state before control is transferred to a function that doesn't have `fenv_access` enabled. This example demonstrates how the `float_control` pragma can be set and restored:

C++

```
#pragma float_control(precise, on, push)
// Code that uses /fp:strict mode
#pragma float_control(pop)
```

Floating-point rounding modes

Under both **`/fp:precise`** and **`/fp:fast`**, the compiler generates code intended to run in the default floating-point environment. It assumes that the environment isn't accessed or modified at runtime. That is, the compiler assumes the code never unmasks floating-point exceptions, reads or writes floating-point status registers, or changes rounding modes. However, some programs need to alter the floating-point environment. For example, this sample computes error bounds of a floating-point multiplication by altering floating-point rounding modes:

C++

```
// fp_error_bounds.cpp
#include <iostream>
#include <limits>
using namespace std;

int main(void)
{
    float a = std::<float>::max();
    float b = -1.1;
    float cLower = 0.0;
    float cUpper = 0.0;
    unsigned int control_word = 0;
    int err = 0;

    // compute lower error bound.
    // set rounding mode to -infinity.
    err = _controlfp_s(&control_word, _RC_DOWN, _MCW_RC);
    if (err)
    {
        cout << "_controlfp_s(&control_word, _RC_DOWN, _MCW_RC) failed with
error:" << err << endl;
    }
    cLower = a * b;

    // compute upper error bound.
    // set rounding mode to +infinity.
```

```
    err = _controlfp_s(&control_word, _RC_UP, _MCW_RC);
    if (err)
    {
        cout << "_controlfp_s(&control_word, _RC_UP, _MCW_RC) failed with
error:" << err << endl;
    }
    cUpper = a * b;

    // restore default rounding mode.
    err = _controlfp_s(&control_word, _CW_DEFAULT, _MCW_RC);
    if (err)
    {
        cout << "_controlfp_s(&control_word, _CW_DEFAULT, _MCW_RC) failed
with error:" << err << endl;
    }
    // display error bounds.
    cout << "cLower = " << cLower << endl;
    cout << "cUpper = " << cUpper << endl;
    return 0;
}
```

Since the compiler assumes the default floating point environment under **`/fp:fast`** and **`/fp:precise`**, it's free to ignore the calls to `_controlfp_s`. For example, when compiled by using both **`/O2`** and **`/fp:precise`** for the x86 architecture, the bounds aren't computed, and the sample program outputs:

Output

```
cLower = -inf
cUpper = -inf
```

When compiled by using both **`/O2`** and **`/fp:strict`** for the x86 architecture, the sample program outputs:

Output

```
cLower = -inf
cUpper = -3.40282e+38
```

Floating-point special values

Under **`/fp:precise`** and **`/fp:strict`**, expressions that involve special values (NaN, +infinity, -infinity, -0.0) behave according to the IEEE-754 specifications. Under **`/fp:fast`**, the behavior of these special values may be inconsistent with IEEE-754.

This sample demonstrates the different behavior of special values under `/fp:precise`, `/fp:strict`, and `/fp:fast`:

C++

```
// fp_special_values.cpp
#include <stdio.h>
#include <cmath>

float gf0 = -0.0;

int main()
{
    float f1 = INFINITY;
    float f2 = NAN;
    float f3 = -INFINITY;
    bool a, b;
    float c, d, e;
    a = (f1 == f1);
    b = (f2 == f2);
    c = (f1 - f1);
    d = (f2 - f2);
    e = (gf0 / f3);
    printf("INFINITY == INFINITY : %d\n", a);
    printf("NAN == NAN           : %d\n", b);
    printf("INFINITY - INFINITY  : %f\n", c);
    printf("NAN - NAN            : %f\n", d);
    printf("std::signbit(-0.0/-INFINITY): %d\n", std::signbit(e));
    return 0;
}
```

When compiled by using `/O2 /fp:precise` or `/O2 /fp:strict` for x86 architecture, the outputs are consistent with the IEEE-754 specification:

Output

```
INFINITY == INFINITY : 1
NAN == NAN           : 0
INFINITY - INFINITY  : -nan(ind)
NAN - NAN            : nan
std::signbit(-0.0/-INFINITY): 0
```

When compiled by using `/O2 /fp:fast`** for x86 architecture, the outputs aren't consistent with IEEE-754:

Output

```
INFINITY == INFINITY : 1
NAN == NAN           : 1
```

```
INFINITY - INFINITY  : 0.000000
NAN - NAN            : 0.000000
std::signbit(-0.0/-INFINITY): 0
```

Floating-point algebraic transformations

Under `/fp:precise` and `/fp:strict`, the compiler doesn't do any mathematical transformation unless the transformation is guaranteed to produce a bitwise identical result. The compiler may make such transformations under `/fp:fast`. For example, the expression `a * b + a * c` in the sample function `algebraic_transformation` may be compiled into `a * (b + c)` under `/fp:fast`. Such transformations aren't done under `/fp:precise` or `/fp:strict`, and the compiler generates `a * b + a * c`.

C++

```
float algebraic_transformation (float a, float b, float c)
{
    return a * b + a * c;
}
```

Floating-point explicit casting points

Under `/fp:precise` and `/fp:strict`, the compiler rounds to source code precision at four specific points during expression evaluation: at assignments, typecasts, when floating-point arguments get passed to a function call, and when a function call returns a floating-point value. Typecasts can be used to explicitly round intermediate computations. Under `/fp:fast`, the compiler doesn't generate explicit casts at these points to guarantee source code precision. This sample demonstrates the behavior under different `/fp` options:

C++

```
float casting(float a, float b)
{
    return 5.0*((double)(a+b));
}
```

When compiled by using `/O2 /fp:precise` or `/O2 /fp:strict`, you can see that explicit type casts are inserted at both the typecast and at the function return point in the generated code for the x64 architecture:

asm

```
        addss    xmm0, xmm1
        cvtss2sd xmm0, xmm0
        mulsd    xmm0, QWORD PTR __real@4014000000000000
        cvtsd2ss xmm0, xmm0
        ret      0
```

Under `/O2 /fp:fast` the generated code is simplified, because all type casts are optimized away:

asm

```
        addss    xmm0, xmm1
        mulss    xmm0, DWORD PTR __real@40a00000
        ret      0
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify the **Floating Point Model** property.

To set this compiler option programmatically

- See floatingPointModel.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/fpcvt` (Floating-point to integer conversion compatibility)

Article • 12/01/2021

Specifies how the compiler treats floating-point conversions to integer types.

Syntax

> `/fpcvt:IA`
>
> `/fpcvt:BC`

Arguments

`/fpcvt:IA`

The `/fpcvt:IA` option tells the compiler to convert floating point values to integers so the results are compatible with the Intel AVX-512 conversion instructions. This behavior is the usual behavior in Visual Studio 2019 for x86 targets.

`/fpcvt:BC`

The `/fpcvt:BC` option tells the compiler to convert floating point values to unsigned integers so the results are compatible with the Visual Studio 2017 and earlier compilers. This behavior is the default in Visual Studio 2022.

Remarks

In Visual Studio 2019 version 16.8 and later versions, the `/fpcvt` compiler option can be used to control the results of floating-point to integer conversions. The `/fpcvt:BC` option specifies the default behavior of Visual Studio 2022, which is the same as the behavior of Visual Studio 2017 and earlier versions. The `/fpcvt:IA` option specifies behavior compatible with Intel Architecture (IA) AVX-512 conversion instruction behavior. This option can be used with either 32-bit x86 or 64-bit x64 targets, and it applies whether /arch:AVX512 is specified or not.

For Visual Studio 2019, the default behavior for x64 targets is consistent with `/fpcvt:BC` unless `/arch:AVX512` is specified. Usually, the behavior for x86 targets is consistent with

`/fpcvt:IA`, except under `/arch:IA32`, `/arch:SSE`, or sometimes where the result of a function call is directly converted to an unsigned integer. Use of `/fpcvt` overrides the default, so all conversions are handled consistently on either target. The behavior of conversions for ARM and ARM64 targets isn't consistent with either `/fpcvt:BC` or `/fpcvt:IA`.

Standard C++ specifies that if a truncated floating-point value is exactly representable in an integer type, it must have that value when converted to that type. Otherwise, any behavior at all is allowed. Both `/fpcvt` options conform with Standard C++. The only difference is in what values are returned for invalid source values.

The `/fpcvt:IA` option causes any invalid conversion to return a single *sentinel* value, which is the destination value farthest from zero. For conversion to signed types, the sentinel is the minimum value for that type. Unsigned types use the maximum value. Floating-point operations may return a Not-a-Number (NaN) value to indicate an invalid operation. That indicator isn't an option for conversion to integer types, which don't have NaN values. The sentinel is used as a proxy for a NaN value, although it can also be the result of a valid conversion.

The `/fpcvt:BC` option also makes conversion to signed types return the minimum possible value when the source is invalid. However, conversion to unsigned integer types is based on conversion to `long long`. To convert a value to `unsigned int`, the compiler first converts it to type `long long`. The compiler then truncates the result to 32 bits. To convert a value to `unsigned long long`, valid source values that are too high for a `long long` are handled as a special case. All other values are first converted to `long long` and then recast to `unsigned long long`.

The `/fpcvt` options are new in Visual Studio 2019 version 16.8. If you specify more than one `/fpcvt` option on the command line, the later option takes precedence and the compiler generates a warning.

Intrinsic functions for conversions

You can specify the behavior of a specific conversion independently of the `/fpcvt` option, which applies globally. The compiler provides intrinsic sentinel conversion functions for conversions compatible with `/fpcvt:IA`. For more information, see Sentinel conversion functions. The compiler also provides saturation conversion functions compatible with conversions on ARM or ARM64 target architectures. For more information, see Saturation conversion functions.

The compiler also supports intrinsic conversion functions that execute as quickly as possible for valid conversions. These functions may generate any value or throw an exception for an invalid conversion. The results depend on the target platform, compiler options, and context. They're useful for handling values that have already been range-checked, or values generated in a way that can't cause an invalid conversion. For more information, see Fast conversion functions.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to add `/fpcvt:IA` or `/fpcvt:BC`. Choose **OK** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax
Fast conversion functions
Saturation conversion functions
Sentinel conversion functions

/FS (Force Synchronous PDB Writes)

Article • 08/03/2021

Forces writes to the program database (PDB) file—created by /Zi or /ZI—to be serialized through MSPDBSRV.EXE.

Syntax

```
/FS
```

Remarks

By default, when **/Zi** or **/ZI** is specified, the compiler locks PDB files to write type information and symbolic debugging information. This can significantly reduce the time it takes the compiler to generate type information when the number of types is large. If another process temporarily locks the PDB file—for example, an anti-virus program—writes by the compiler may fail and a fatal error may occur. This problem can also happen when multiple copies of cl.exe access the same PDB file—for example, if your solution has independent projects that use the same intermediate directories or output directories and parallel builds are enabled. The **/FS** compiler option prevents the compiler from locking the PDB file and forces writes to go through MSPDBSRV.EXE, which serializes access. This may make builds significantly longer, and it doesn't prevent all errors that may occur when multiple instances of cl.exe access the PDB file at the same time. We recommend that you change your solution so that independent projects write to separate intermediate and output locations, or that you make one of the projects dependent on the other to force serialized project builds.

The /MP option enables **/FS** by default.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

3. Modify the **Additional Options** property to include `/FS` and then choose **OK**.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/fsanitize` (Enable sanitizers)

Article • 02/20/2025

Use the `/fsanitize` compiler options to enable sanitizers.

Syntax

> `/fsanitize=address`
>
> `/fsanitize=kernel-address`
>
> `/fsanitize=fuzzer`
>
> `/fsanitize-address-use-after-return`
>
> `/fno-sanitize-address-vcasan-lib` `/fsanitize-address-asan-compat-lib` `/fno-sanitize-address-asan-compat-lib`

Remarks

The `/fsanitize=address` compiler option enables AddressSanitizer, a powerful compiler and runtime technology to uncover hard-to-find bugs. Support for the `/fsanitize=address` option is available starting in Visual Studio 2019 version 16.9.

The `/fsanitize=kernel-address` compiler option enables Kernel AddressSanitizer (KASan). KASan is the kernel-mode variant of AddressSanitizer, available starting in Visual Studio 2022 version 17.11. KASan is only supported on Windows 11 24H2 or Windows Server 2025 and higher, and requires building using a Windows SDK 10.0.26100.2161 and higher. Building with KASan also implies the `/fsanitize-address-asan-compat-lib` compiler option.

The `/fsanitize=fuzzer` compiler option enables experimental support for LibFuzzer. LibFuzzer is a coverage-guided fuzzing library that can be used to find bugs and crashes caused by user-provided input. We recommended you use `/fsanitize=address` with LibFuzzer. This option is useful for fuzzing tools such as OneFuzz. For more information, see the OneFuzz documentation and OneFuzz GitHub project. Support for the `/fsanitize=fuzzer` option is available starting in Visual Studio 2022 version 17.0.

The `/fsanitize` option doesn't allow comma-separated syntax, for example: `/fsanitize=address,fuzzer`. These options must be specified individually.

The `/fsanitize-address-use-after-return`, `/fno-sanitize-address-vcasan-lib`, `/fsanitize-address-asan-compat-lib`, and `/fno-sanitize-address-asan-compat-lib`

compiler options, and the /INFERASANLIBS (Use inferred sanitizer libs) and `/INFERASANLIBS:NO` linker options offer support for advanced users. For more information, see AddressSanitizer build and language reference.

To set the `/fsanitize=address` compiler option in the Visual Studio development environment

1. Open your project's **Property Pages** dialog box.
2. Select the **Configuration Properties** > **C/C++** > **General** property page.
3. Modify the **Enable Address Sanitizer** property. To enable it, choose **Yes (/fsanitize=address)**.
4. Choose **OK** or **Apply** to save your changes.

To set the `/fsanitize=fuzzer` compiler option in the Visual Studio development environment

1. Open your project's **Property Pages** dialog box.
2. Select the **Configuration Properties** > **C/C++** > **General** property page.
3. Modify the **Enable Fuzzer** property. To enable it, choose **Yes (/fsanitize=fuzzer)**.
4. Choose **OK** or **Apply** to save your changes.

To set the advanced compiler options

1. Open your project's **Property Pages** dialog box.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to set **/fsanitize-address-use-after-return** or **/fno-sanitize-address-vcasan-lib**.
4. Choose **OK** or **Apply** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax
/INFERASANLIBS (Use inferred sanitizer libs)
/fsanitize-coverage (Configure sanitizer coverage)
AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/fsanitize-coverage` (Configure sanitizer coverage)

Article • 11/04/2021

The `/fsanitize-coverage` compiler options instruct the compiler to add various kinds of instrumentation points where user-defined functions are called. These options are useful for fuzzing scenarios that use `/fsanitize=fuzzer`, like OneFuzz. For more information, see the OneFuzz documentation and OneFuzz GitHub project.

Syntax

> `/fsanitize-coverage=edge`
>
> `/fsanitize-coverage=inline-8bit-counters`
>
> `/fsanitize-coverage=trace-cmp`
>
> `/fsanitize-coverage=trace-div`
>
> `/fno-sanitize-coverage=edge`
>
> `/fno-sanitize-coverage=inline-8bit-counters`
>
> `/fno-sanitize-coverage=trace-cmp`
>
> `/fno-sanitize-coverage=trace-div`

Remarks

The experimental `/fsanitize-coverage` compiler options offer code coverage support and various options to modify which compiler-provided instrumentation is generated. All these options are automatically set when the /fsanitize=fuzzer option is specified. The `/fsanitize=fuzzer` option requires the same instrumentation points and callbacks mentioned in these options.

The `/fsanitize-coverage` options don't allow comma-separated syntax, for example: `/fsanitize-coverage=edge,inline-8bit-counters,trace-cmp,trace-div`. Specify these options individually.

The `/fsanitize-coverage` options are available beginning in Visual Studio 2022 version 17.0.

Code coverage

The `/fsanitize-coverage=edge` compiler option enables code coverage instrumentation along all non-redundant edges. Use `/fno-sanitize-coverage=edge` to disable this option if it's already provided or implied by another option.

Inline counters

The `/fsanitize-coverage=inline-8bit-counters` compiler option instructs the compiler to add an inline counter increment on every relevant edge. This option also adds a call to `extern "C" void __sanitizer_cov_8bit_counters_init(uint8_t *start, uint8_t *stop)` that you must implement. The arguments correspond to the start and end of an array that contains all the 8-bit counters created. Use `/fno-sanitize-coverage=inline-8bit-counters` to disable this option if it's already provided or implied by another option.

Trace comparisons

The `/fsanitize-coverage=trace-cmp` compiler option instructs the compiler to insert calls to the following functions:

C

```
// Before each comparison instruction of the stated size.
void __sanitizer_cov_trace_cmp1(uint8_t Arg1, uint8_t Arg2);
void __sanitizer_cov_trace_cmp2(uint16_t Arg1, uint16_t Arg2);
void __sanitizer_cov_trace_cmp4(uint32_t Arg1, uint32_t Arg2);
void __sanitizer_cov_trace_cmp8(uint64_t Arg1, uint64_t Arg2);

// Before each comparison instruction of the stated size, if one of the
operands (Arg1) is constant.
void __sanitizer_cov_trace_const_cmp1(uint8_t Arg1, uint8_t Arg2);
void __sanitizer_cov_trace_const_cmp2(uint16_t Arg1, uint16_t Arg2);
void __sanitizer_cov_trace_const_cmp4(uint32_t Arg1, uint32_t Arg2);
void __sanitizer_cov_trace_const_cmp8(uint64_t Arg1, uint64_t Arg2);
```

Use `/fno-sanitize-coverage=trace-cmp` to disable this option if it's already provided or implied by another option.

Trace divisions

The `/fsanitize-coverage=trace-div` compiler option instructs the compiler to insert calls to the following functions:

C

```
// Before a division instruction of the stated size.
void __sanitizer_cov_trace_div4(uint32_t Val);
void __sanitizer_cov_trace_div8(uint64_t Val);
```

Use `/fno-sanitize-coverage=trace-div` to disable this option if it's already provided or implied by another option.

To set the advanced compiler options

1. Open your project's **Property Pages** dialog box.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to set **/fsanitize-coverage** options.
4. Choose **OK** or **Apply** to save your changes.

See also

MSVC compiler options
MSVC compiler command-line syntax
/fsanitize (Enable Sanitizers)
AddressSanitizer build and language reference

/GA (Optimize for Windows Application)

Article • 08/03/2021

Results in more efficient code for an .exe file for accessing thread-local storage (TLS) variables.

Syntax

```
/GA
```

Remarks

/GA speeds access to data declared with __declspec(thread) in a Windows-based program. When this option is set, the __tls_index macro is assumed to be 0.

Using **/GA** for a DLL can result in bad code generation.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Gd, /Gr, /Gv, /Gz (Calling Convention)

Article • 08/03/2021

These options determine the order in which function arguments are pushed onto the stack, whether the caller function or called function removes the arguments from the stack at the end of the call, and the name-decorating convention that the compiler uses to identify individual functions.

Syntax

> `/Gd`
>
> `/Gr`
>
> `/Gv`
>
> `/Gz`

Remarks

`/Gd`, the default setting, specifies the __cdecl calling convention for all functions except C++ member functions and functions that are marked __stdcall, __fastcall, or __vectorcall.

`/Gr` specifies the `__fastcall` calling convention for all functions except C++ member functions, functions named `main`, and functions that are marked `__cdecl`, `__stdcall`, or `__vectorcall`. All `__fastcall` functions must have prototypes. This calling convention is only available in compilers that target x86, and is ignored by compilers that target other architectures.

`/Gz` specifies the `__stdcall` calling convention for all functions except C++ member functions, functions named `main`, and functions that are marked `__cdecl`, `__fastcall`, or `__vectorcall`. All `__stdcall` functions must have prototypes. This calling convention is only available in compilers that target x86, and is ignored by compilers that target other architectures.

`/Gv` specifies the `__vectorcall` calling convention for all functions except C++ member functions, functions named `main`, functions with a `vararg` variable argument list, or functions that are marked with a conflicting `__cdecl`, `__stdcall`, or `__fastcall` attribute. This calling convention is only available on x86 and x64 architectures that support /arch:SSE2 and above, and is ignored by compilers that target the ARM architecture.

Functions that take a variable number of arguments must be marked `__cdecl`.

`/Gd`, `/Gr`, `/Gv` and `/Gz` are not compatible with /clr:safe or **/clr:pure**. The **/clr:pure** and **/clr:safe** compiler options are deprecated in Visual Studio 2015 and unsupported in Visual Studio 2017 and later.

> **Note**
>
> By default for x86 processors, C++ member functions use **__thiscall**.

For all processors, a member function that is explicitly marked as `__cdecl`, `__fastcall`, `__vectorcall`, or `__stdcall` uses the specified calling convention if it is not ignored on that architecture. A member function that takes a variable number of arguments always uses the `__cdecl` calling convention.

These compiler options have no effect on the name decoration of C++ methods and functions. Unless declared as `extern "C"`, C++ methods and functions use a different name-decorating scheme. For more information, see Decorated Names.

For more information about calling conventions, see Calling Conventions.

__cdecl Specifics

On x86 processors, all function arguments are passed on the stack from right to left. On ARM and x64 architectures, some arguments are passed by register and the rest are passed on the stack from right to left. The calling routine pops the arguments from the stack.

For C, the `__cdecl` naming convention uses the function name preceded by an underscore (`_`); no case translation is performed. Unless declared as `extern "C"`, C++ functions use a different name-decorating scheme. For more information, see Decorated Names.

__fastcall Specifics

Some of a `__fastcall` function's arguments are passed in registers (for x86 processors, ECX, and EDX), and the rest are pushed onto the stack from right to left. The called routine pops these arguments from the stack before it returns. Typically, **/Gr** decreases execution time.

> ⓘ Note
>
> Be careful when you use the `__fastcall` calling convention for any function that's written in inline assembly language. Your use of registers could conflict with the compiler's use.

For C, the `__fastcall` naming convention uses the function name preceded by an at sign (@) followed by the size of the function's arguments in bytes. No case translation is done. The compiler uses this template for the naming convention:

```
@function_name@number
```

When you use the `__fastcall` naming convention, use the standard include files. Otherwise, you will get unresolved external references.

__stdcall Specifics

A `__stdcall` function's arguments are pushed onto the stack from right to left, and the called function pops these arguments from the stack before it returns.

For C, the `__stdcall` naming convention uses the function name preceded by an underscore (_) and followed by an at sign (@) and the size of the function's arguments in bytes. No case translation is performed. The compiler uses this template for the naming convention:

```
_functionname@number
```

__vectorcall Specifics

A `__vectorcall` function's integer arguments are passed by value, using up to two (on x86) or four (on x64) integer registers, and up to six XMM registers for floating-point and vector values, and the rest are passed on the stack from right to left. The called function cleans off the stack before it returns. Vector and floating-point return values are returned in XMM0.

For C, the `__vectorcall` naming convention uses the function name followed by two at signs (@@) and the size of the function's arguments in bytes. No case translation is performed. The compiler uses this template for the naming convention:

```
functionname@@number
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Advanced** property page.
3. Modify the **Calling Convention** property.

To set this compiler option programmatically

- See CallingConvention.

See also

- MSVC Compiler Options
- MSVC Compiler Command-Line Syntax

/Ge (Enable Stack Probes)

Article • 08/03/2021

Activates stack probes for every function call that requires storage for local variables.

Syntax

```
/Ge
```

Remarks

This mechanism is useful if you rewrite the functionality of the stack probe. It is recommended that you use /Gh (Enable _penter Hook Function) instead of rewriting the stack probe.

/Gs (Control Stack Checking Calls) has the same effect.

/Ge is deprecated; beginning in Visual Studio 2005, the compiler automatically generates stack checking. For a list of deprecated compiler options, see **Deprecated and Removed Compiler Options** in Compiler Options Listed by Category.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options

MSVC Compiler Command-Line Syntax

/GF (Eliminate Duplicate Strings)

Article • 08/03/2021

Enables the compiler to create a single copy of identical strings in the program image and in memory during execution. This is an optimization called *string pooling* that can create smaller programs.

Syntax

```
/GF
```

Remarks

If you use **/GF**, the operating system does not swap the string portion of memory and can read the strings back from the image file.

/GF pools strings as read-only. If you try to modify strings under **/GF**, an application error occurs.

String pooling allows what were intended as multiple pointers to multiple buffers to be multiple pointers to a single buffer. In the following code, `s` and `t` are initialized with the same string. String pooling causes them to point to the same memory:

```
char *s = "This is a character buffer";
char *t = "This is a character buffer";
```

> **Note**
>
> The **/ZI** option, used for Edit and Continue, automatically sets the **/GF** option.

> **Note**
>
> The **/GF** compiler option creates an addressable section for each unique string. And by default, an object file can contain up to 65,536 addressable sections. If your

program contains more than 65,536 strings, use the **/bigobj** compiler option to create more sections.

/GF is in effect when /O1 or /O2 is used.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify the **Enable String Pooling** property.

To set this compiler option programmatically

- See StringPooling.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/GH (Enable _pexit hook function)

Article • 08/03/2021

Calls the `_pexit` function at the end of every method or function.

Syntax

> `/GH`

Remarks

The `_pexit` function isn't part of any library. It's up to you to provide a definition for `_pexit`.

Unless you plan to explicitly call `_pexit`, you don't need to provide a prototype. The function must push the content of all registers on entry and pop the unchanged content on exit. It must appear as if it had the following prototype:

```
C++

void __declspec(naked) __cdecl _pexit( void );
```

This declaration isn't available for 64-bit projects.

`_pexit` is similar to `_penter`; see /Gh (Enable _penter Hook Function) for an example of how to write a `_penter` function.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax
/Gh (Enable _penter hook function)

/Gh (Enable _penter hook function)

Article • 06/30/2022

Causes a call to the `_penter` function at the start of every method or function.

Syntax

> `/Gh`

Remarks

The `_penter` function isn't part of any library. It's up to you to provide a definition for `_penter`.

Unless you plan to explicitly call `_penter`, you don't need to provide a prototype. The function must push the content of all registers on entry and pop the unchanged content on exit. It must appear as if it had the following prototype:

```
C++

void __declspec(naked) __cdecl _penter( void );
```

This declaration isn't available for 64-bit projects.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

Example

The following code, when compiled with **/Gh**, shows how `_penter` is called twice; once when entering function `main` and once when entering function `x`. The example consists of two source files, which you compile separately.

Source file `local_penter.cpp`:

C++

```
// local_penter.cpp
// compile with: cl /EHsc /c local_penter.cpp
// processor: x86
#include <stdio.h>

extern "C" void __declspec(naked) __cdecl _penter( void ) {
   _asm {
      push eax
      push ebx
      push ecx
      push edx
      push ebp
      push edi
      push esi
    }

   printf_s("\nIn a function!");

   _asm {
      pop esi
      pop edi
      pop ebp
      pop edx
      pop ecx
      pop ebx
      pop eax
      ret
    }
}
```

Source file `Gh_compiler_option.cpp`:

C++

```
// Gh_compiler_option.cpp
// compile with: cl /EHsc /Gh Gh_compiler_option.cpp local_penter.obj
// processor: x86
#include <stdio.h>

void x() {}

int main() {
```

```
    x();
}
```

When run, the local `_penter` function is called on entry to `main` and `x`:

Output

```
In a function!
In a function!
```

See also

MSVC compiler options
MSVC compiler command-line syntax
/GH (Enable _pexit hook function)

`/GL` (Whole program optimization)

Article • 08/03/2021

Enables whole program optimization.

Syntax

> `/GL`[`-`]

Remarks

Whole program optimization allows the compiler to perform optimizations with information on all modules in the program. Without whole program optimization, optimizations are performed on a per-module (compiland) basis.

Whole program optimization is off by default and must be explicitly enabled. However, it's also possible to explicitly disable it with `/GL-`.

With information on all modules, the compiler can:

- Optimize the use of registers across function boundaries.
- Do a better job of tracking modifications to global data, allowing a reduction in the number of loads and stores.
- Track the possible set of items modified by a pointer dereference, reducing the required loads and stores.
- Inline a function in a module even when the function is defined in another module.

`.obj` files produced with `/GL` aren't usable by linker utilities such as EDITBIN and DUMPBIN.

If you compile your program with `/GL` and /c, you should use the /LTCG linker option to create the output file.

/ZI can't be used with `/GL`

The format of files produced with `/GL` in the current version often isn't readable by later versions of Visual Studio and the MSVC toolset. Unless you're willing to ship copies of the `.lib` file for all versions of Visual Studio you expect your users to use, now and in

the future, don't ship a `.lib` file made up of `.obj` files produced by `/GL` . For more information, see Restrictions on binary compatibility.

`.obj` files produced by `/GL` and precompiled header files shouldn't be used to build a `.lib` file unless the `.lib` file is linked on the same machine that produced the `/GL` `.obj` file. Information from the `.obj` file's precompiled header file is needed at link time.

For more information on the optimizations available with and the limitations of whole program optimization, see /LTCG. `/GL` also makes profile guided optimization available. When compiling for profile guided optimizations and if you want function ordering from your profile guided optimizations, you must compile with /Gy or a compiler option that implies /Gy.

To set this linker option in the Visual Studio development environment

For more information on how to specify `/GL` in the development environment, see /LTCG (Link-time code generation) .

To set this linker option programmatically

- See WholeProgramOptimization.

See also

MSVC compiler options
MSVC compiler command-line syntax

/Gm (Enable Minimal Rebuild)

Article • 08/03/2021

Deprecated. Enables minimal rebuild, which determines whether C++ source files that include changed C++ class definitions (stored in header (.h) files) need to be recompiled.

Syntax

```
/Gm
```

Remarks

/Gm is deprecated. It may not trigger a build for certain kinds of header file changes. You may safely remove this option from your projects. To improve build times, we recommend you use precompiled headers and incremental and parallel build options instead. For a list of deprecated compiler options, see the **Deprecated and Removed Compiler Options** section in Compiler Options Listed by Category.

The compiler stores dependency information between source files and class definitions in the project's .idb file during the first compile. (Dependency information tells which source file is dependent on which class definition, and which .h file the definition is located in.) Subsequent compiles use the information stored in the .idb file to determine whether a source file needs to be compiled, even if it includes a modified .h file.

> **Note**
>
> Minimal rebuild relies on class definitions not changing between include files. Class definitions must be global for a project (there should be only one definition of a given class), because the dependency information in the .idb file is created for the entire project. If you have more than one definition for a class in your project, disable minimal rebuild.

Because the incremental linker does not support the Windows metadata included in .obj files by using the /ZW (Windows Runtime Compilation) option, the **/Gm** option is incompatible with **/ZW**.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify the **Enable Minimal Rebuild** property.

To set this compiler option programmatically

- See MinimalRebuild.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/GR (Enable Run-Time Type Information)

Article • 08/03/2021

Adds code to check object types at run time.

Syntax

```
/GR[-]
```

Remarks

When **/GR** is on, the compiler defines the `_CPPRTTI` preprocessor macro. By default, **/GR** is on. **/GR-** disables run-time type information.

Use **/GR** if the compiler cannot statically resolve an object type in your code. You usually need the **/GR** option when your code uses dynamic_cast Operator or typeid. However, **/GR** increases the size of the .rdata sections of your image. If your code does not use `dynamic_cast` or `typeid`, **/GR-** may produce a smaller image.

For more information about run-time type checking, see Run-Time Type Information in the *C++ Language Reference*.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Language** property page.
3. Modify the **Enable Run-Time Type Info** property.

To set this compiler option programmatically

- See RuntimeTypeInfo.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/GS (Buffer Security Check)

Article • 08/03/2021

Detects some buffer overruns that overwrite a function's return address, exception handler address, or certain types of parameters. Causing a buffer overrun is a technique used by hackers to exploit code that does not enforce buffer size restrictions.

Syntax

```
/GS[-]
```

Remarks

/GS is on by default. If you expect your application to have no security exposure, use **/GS-**. For more information about suppressing buffer overrun detection, see safebuffers.

Security Checks

On functions that the compiler recognizes as subject to buffer overrun problems, the compiler allocates space on the stack before the return address. On function entry, the allocated space is loaded with a *security cookie* that is computed once at module load. On function exit, and during frame unwinding on 64-bit operating systems, a helper function is called to make sure that the value of the cookie is still the same. A different value indicates that an overwrite of the stack may have occurred. If a different value is detected, the process is terminated.

GS Buffers

A buffer overrun security check is performed on a *GS buffer*. A GS buffer can be one of these:

- An array that is larger than 4 bytes, has more than two elements, and has an element type that is not a pointer type.
- A data structure whose size is more than 8 bytes and contains no pointers.
- A buffer allocated by using the _alloca function.

- Any class or structure that contains a GS buffer.

For example, the following statements declare GS buffers.

```
C++

char buffer[20];
int buffer[20];
struct { int a; int b; int c; int d; } myStruct;
struct { int a; char buf[20]; };
```

However, the following statements do not declare GS buffers. The first two declarations contain elements of pointer type. The third and fourth statements declare arrays whose size is too small. The fifth statement declares a structure whose size on an x86 platform is not more than 8 bytes.

```
C++

char *pBuf[20];
void *pv[20];
char buf[4];
int buf[2];
struct { int a; int b; };
```

Initialize the Security Cookie

The **/GS** compiler option requires that the security cookie be initialized before any function that uses the cookie is run. The security cookie must be initialized immediately on entry to an EXE or DLL. This is done automatically if you use the default VCRuntime entry points: mainCRTStartup, wmainCRTStartup, WinMainCRTStartup, wWinMainCRTStartup, or _DllMainCRTStartup. If you use an alternate entry point, you must manually initialize the security cookie by calling __security_init_cookie.

What Is Protected

The **/GS** compiler option protects the following items:

- The return address of a function call.
- The address of an exception handler for a function.
- Vulnerable function parameters.

On all platforms, **/GS** attempts to detect buffer overruns into the return address. Buffer overruns are more easily exploited on platforms such as x86 and x64, which use calling conventions that store the return address of a function call on the stack.

On x86, if a function uses an exception handler, the compiler injects a security cookie to protect the address of the exception handler. The cookie is checked during frame unwinding.

/GS protects *vulnerable parameters* that are passed into a function. A vulnerable parameter is a pointer, a C++ reference, a C-structure (C++ POD type) that contains a pointer, or a GS buffer.

A vulnerable parameter is allocated before the cookie and local variables. A buffer overrun can overwrite these parameters. And code in the function that uses these parameters could cause an attack before the function returns and the security check is performed. To minimize this danger, the compiler makes a copy of the vulnerable parameters during the function prolog and puts them below the storage area for any buffers.

The compiler does not make copies of vulnerable parameters in the following situations:

- Functions that do not contain a GS buffer.
- Optimizations (/O options) are not enabled.
- Functions that have a variable argument list (...).
- Functions that are marked with naked.
- Functions that contain inline assembly code in the first statement.
- A parameter is used only in ways that are less likely to be exploitable in the event of a buffer overrun.

What Is Not Protected

The **/GS** compiler option does not protect against all buffer overrun security attacks. For example, if you have a buffer and a vtable in an object, a buffer overrun could corrupt the vtable.

Even if you use **/GS**, always try to write secure code that has no buffer overruns.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify the **Buffer Security Check** property.

To set this compiler option programmatically

- See BufferSecurityCheck.

Example

This sample overruns a buffer. This causes the application to fail at runtime.

C

```
// compile with: /c /W1
#include <cstring>
#include <stdlib.h>
#pragma warning(disable : 4996)   // for strcpy use

// Vulnerable function
void vulnerable(const char *str) {
   char buffer[10];
   strcpy(buffer, str); // overrun buffer !!!

   // use a secure CRT function to help prevent buffer overruns
   // truncate string to fit a 10 byte buffer
   // strncpy_s(buffer, _countof(buffer), str, _TRUNCATE);
}

int main() {
   // declare buffer that is bigger than expected
   char large_buffer[] = "This string is longer than 10 characters!!";
   vulnerable(large_buffer);
}
```

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/Gs` (Control stack checking calls)

Article • 02/17/2023

Controls the threshold for stack probes.

Syntax

> **`/Gs`**[*`size`*]

Arguments

`size`
(Optional) The number of bytes that local variables can occupy before a stack probe is initiated. No whitespace is allowed between **`/Gs`** and *`size`*.

Remarks

A *stack probe* is a sequence of code that the compiler inserts at the beginning of a function call. When initiated, a stack probe reaches benignly into memory by the amount of space required to store the function's local variables. This probe causes the operating system to transparently page in more stack memory if necessary, before the rest of the function runs.

By default, the compiler generates code that initiates a stack probe when a function requires more than one page of stack space. This default is equivalent to a compiler option of **`/Gs4096`** for x86, x64, ARM, and ARM64 platforms. This value allows an application and the Windows memory manager to increase the amount of memory committed to the program stack dynamically at run time.

> ⓘ **Note**
>
> The default value of **`/Gs4096`** allows the program stack of applications for Windows to grow correctly at run time. We recommend that you do not change the default value unless you know exactly why you have to change it.

Some programs—for example, virtual device drivers—don't require this default stack-growth mechanism. In such cases, the stack probes aren't necessary and you can stop

the compiler from generating them by setting *size* to a value that is larger than any function requires for local variable storage.

/Gs0 initiates stack probes for every function call that requires storage for local variables. This value can have a negative impact on performance.

For x64 targets, if you specify the **/Gs** option without a *size* argument, it's the same as **/Gs0**. If the *size* argument is 1 through 9, the compiler emits warning D9014, and the effect is the same as specifying **/Gs0**.

For x86, ARM, and ARM64 targets, the **/Gs** option without a *size* argument is the same as **/Gs4096**. If the *size* argument is 1 through 9, the compiler emits warning D9014, and the effect is the same as specifying **/Gs4096**.

For all targets, a *size* argument between 10 and 2147483647 sets the threshold at the specified value. A *size* of 2147483648 or greater causes fatal error C1049.

You can turn stack probes on or off by using the check_stack directive. **/Gs** and the `check_stack` pragma have no effect on standard C library routines; they affect only the functions you compile.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the **/Gs** compiler option and an optional size in **Additional Options**. Choose **OK** or **Apply** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/guard` (Enable Control Flow Guard)

Article • 02/25/2025

Enable compiler generation of Control Flow Guard security checks.

Syntax

> `/guard:cf`
>
> `/guard:cf-`

Remarks

The `/guard:cf` option causes the compiler to analyze control flow for indirect call targets at compile time, and inserts code at runtime to verify the targets. By default, `/guard:cf` is off and must be explicitly enabled. To explicitly disable this option, use `/guard:cf-`.

Visual Studio 2017 and later: This option adds guards for `switch` statements that generate jump tables.

When the `/guard:cf` Control Flow Guard (CFG) option is specified, the compiler and linker insert extra runtime security checks to detect attempts to compromise your code. During compiling and linking, all indirect calls in your code are analyzed to find every location that the code can reach when it runs correctly. This information is stored in extra structures in the headers of your binaries. The compiler also injects a check before every indirect call in your code that ensures the target is one of the verified locations. If the check fails at runtime on a CFG-aware operating system, the operating system closes the program.

A common attack on software takes advantage of bugs in handling extreme or unexpected inputs. Carefully crafted input to the application may overwrite a location that contains a pointer to executable code. This technique can be used to redirect control flow to code controlled by the attacker. The CFG runtime checks don't fix the data corruption bugs in your executable. They instead make it more difficult for an attacker to use them to execute arbitrary code. CFG is a mitigation tool that prevents calls to locations other than function entry points in your code. It's similar to how Data Execution Prevention (DEP), /GS stack checks, and /DYNAMICBASE and /HIGHENTROPYVA address space layout randomization (ASLR) lower the chances that your code becomes an exploit vector.

To use the CFG exploit mitigation technique, pass `/guard:cf` to the compiler and `/GUARD:CF` to the linker.

To disable the CFG exploit mitigation technique, pass `/guard:cf-` to the compiler `/GUARD:NO` to the linker.

If you build using a single `cl` command, the compiler passes the option to the linker. If you compile and link separately, set the option for both the compiler and linker commands. The `/DYNAMICBASE` linker option is also required.

To verify that your binary has CFG data, use the `dumpbin /headers /loadconfig` command. CFG-enabled binaries have `Guard` in the list of EXE or DLL characteristics, and Guard Flags include `CF Instrumented` and `FID table present`.

The `/guard:cf` option is incompatible with /ZI (Edit and Continue debug information) or /clr (Common Language Runtime Compilation).

Code compiled by using `/guard:cf` can be linked to libraries and object files that aren't compiled by using the option. Only this code, when also linked by using the `/guard:cf` option and run on a CFG-aware operating system, has CFG protection. Because code compiled without the option won't stop an attack, we recommend that you use the option on all the code you compile. There's a small runtime cost for CFG checks, but the compiler analysis attempts to optimize away the checks on indirect jumps that can be proven to be safe.

To set this compiler option in the Visual Studio development environment

1. Open the **Property Pages** dialog box for the project. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Select the **Control Flow Guard** property.
4. In the dropdown control, choose **Yes** to enable Control Flow Guard, or **No** to disable it.

See also

MSVC compiler options
MSVC compiler command-line syntax

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/guard:ehcont` (Enable EH Continuation Metadata)

Article • 08/03/2021

Enables generation of EH Continuation (EHCONT) metadata by the compiler.

Syntax

> **`/guard:ehcont`**[**`-`**]

Remarks

The **`/guard:ehcont`** option causes the compiler to generate a sorted list of the relative virtual addresses (RVA) of all the valid exception handling continuation targets for a binary. It's used during runtime for `NtContinue` and `SetThreadContext` instruction pointer validation. By default, **`/guard:ehcont`** is off and must be explicitly enabled. To explicitly disable this option, use **`/guard:ehcont-`**.

The **`/guard:ehcont`** option is available in Visual Studio 2019 version 16.7 and later. The feature is supported for 64-bit processes on a 64-bit OS.

Control-flow Enforcement Technology (CET) is a hardware-based security feature that protects against Return-Oriented Programming (ROP)-based attacks. It maintains a "shadow stack" for every call stack to enforce control-flow integrity.

When shadow stacks are available to prevent ROP attacks, attackers move on to use other exploit techniques. One technique they may use is to corrupt the instruction pointer value inside the CONTEXT structure. This structure gets passed into system calls that redirect the execution of a thread, such as `NtContinue`, RtlRestoreContext, and SetThreadContext. The `CONTEXT` structure is stored in memory. Corrupting the instruction pointer it contains can cause the system calls to transfer execution to an attacker-controlled address. Currently, `NTContinue` can be called with any continuation point. That's why it's essential to validate the instruction pointer when shadow stacks are enabled.

`RtlRestoreContext` and `NtContinue` are used during Structured Exception Handling (SEH) exception unwinding to unwind to the target frame that contains the **`__except`** block. The instruction pointer of the **`__except`** block isn't expected to be on the shadow

stack, because it would fail instruction pointer validation. The `/guard:ehcont` compiler switch generates an "EH Continuation Table". It contains a sorted list of the RVAs of all valid exception handling continuation targets in the binary. `NtContinue` first checks the shadow stack for the user-supplied instruction pointer, and if the instruction pointer isn't found there, it proceeds to check the EH Continuation Table from the binary that contains the instruction pointer. If the containing binary wasn't compiled with the table, then for compatibility with legacy binaries, `NtContinue` is allowed to continue. It's important to distinguish between legacy binaries that have no EHCONT data, and binaries containing EHCONT data but no table entries. The former allow all addresses inside the binary as valid continuation targets. The latter don't allow any address inside the binary as a valid continuation target.

The `/guard:ehcont` option must be passed to both the compiler and linker to generate EH continuation target RVAs for a binary. If your binary is built by using a single `cl` command, the compiler passes the option to the linker. The compiler also passes the /guard:cf option to the linker. If you compile and link separately, these options must be set on both the compiler and linker commands.

You can link code compiled by using `/guard:ehcont` to libraries and object files compiled without it. The linker returns a fatal error in any of these scenarios:

- A code section has "local unwind". For more information, see Abnormal termination in try-finally Statement.
- An EH (xdata) section contains pointers to a code section, and they aren't for SEH.
- The pointers are for SEH, but the object file wasn't compiled using function-level linking (/Gy) to produce COMDATs.

The linker returns a fatal error, because it can't generate metadata in these scenarios. That means throwing an exception is likely to cause a crash at runtime.

For SEH section info found in COMDATs, but not compiled using `/guard:ehcont`, the linker emits warning **LNK4291**. In this case, the linker generates correct but conservative metadata for the section. To ignore this warning, use /IGNORE (Ignore Specific Warnings).

If the linker is unable to generate metadata, it emits one of the following errors:

- `LNK2046: module contains _local_unwind but was not compiled with /guard:ehcont`

- `LNK2047: module contains C++ EH or complex EH metadata but was not compiled with /guard:ehcont.`

To check if a binary contains EHCONT data, look for the following elements when dumping the binary's load config:

Windows Command Prompt

```
e:\>link /dump /loadconfig CETTest.exe
...
            10417500 Guard Flags
...
                       EH Continuation table present      // EHCONT guard
flag present
...
    0000000180018640 Guard EH continuation table
                  37 Guard EH continuation count          // May be 0 if no
exception handling is used in the binary. Still counts has having EHCONT
data.
...
    Guard EH Continuation Table                           // List of RVAs

          Address
          --------
           0000000180002CF5
           0000000180002F03
           0000000180002F0A
...
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Select the **Enable EH Continuation Metadata** property.
4. In the dropdown control, choose **Yes (/guard:ehcont)** to enable EH continuation metadata, or **No (/guard:ehcont-)** to disable it.

See also

/guard (Enable Control Flow Guard)
MSVC Compiler Options

MSVC Compiler Command-Line Syntax

`/GT` (Support fiber-safe thread-local storage)

Article • 08/03/2021

Supports fiber safety for data allocated using static thread-local storage, that is, data allocated with `__declspec(thread)`.

Syntax

> `/GT`

Remarks

Data declared with `__declspec(thread)` is referenced through a thread-local storage (TLS) array. The TLS array is an array of addresses that the system maintains for each thread. Each address in this array gives the location of thread-local storage data.

A fiber is a lightweight object that consists of a stack and a register context and can be scheduled on various threads. A fiber can run on any thread. Because a fiber may get swapped out and restarted later on a different thread, the compiler mustn't cache the address of the TLS array, or optimize it as a common subexpression across a function call. `/GT` prevents such optimizations.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Optimization** property page.
3. Modify the **Enable Fiber-safe Optimizations** property.

To set this compiler option programmatically

- See EnableFiberSafeOptimizations.

See also

MSVC compiler options

MSVC compiler command-line syntax

/Gw (Optimize Global Data)

Article • 08/03/2021

Package global data in COMDAT sections for optimization.

Syntax

```
/Gw[-]
```

Remarks

The **/Gw** option causes the compiler to package global data in individual COMDAT sections. By default, **/Gw** is off and must be explicitly enabled. To explicitly disable it, use **/Gw-**. When both **/Gw** and /GL are enabled, the linker uses whole-program optimization to compare COMDAT sections across multiple object files in order to exclude unreferenced global data or to merge identical read-only global data. This can significantly reduce the size of the resulting binary executable.

When you compile and link separately, you can use the /OPT:REF linker option to exclude from the executable the unreferenced global data in object files compiled with the **/Gw** option.

You can also use the /OPT:ICF and /LTCG linker options together to merge in the executable any identical read-only global data across multiple object files compiled with the **/Gw** option.

For more information, see Introducing /Gw Compiler Switch on the C++ Team Blog.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include **/Gw** and then choose **OK**.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/GX (Enable Exception Handling)

Article • 08/03/2021

Deprecated. Enables synchronous exception handling using the assumption that functions declared by using `extern "C"` never throw an exception.

Syntax

```
/GX
```

Remarks

/GX is deprecated. Use the equivalent /EHsc option instead. For a list of deprecated compiler options, see the **Deprecated and Removed Compiler Options** section in Compiler Options Listed by Category.

By default, **/EHsc**, the equivalent of **/GX**, is in effect when you compile by using the Visual Studio development environment. When using the command line tools, no exception handling is specified. This is the equivalent of **/GX-**.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
/EH (Exception Handling Model)

/Gy (Enable Function-Level Linking)

Article • 08/03/2021

Allows the compiler to package individual functions in the form of packaged functions (COMDATs).

Syntax

```
/Gy[-]
```

Remarks

The linker requires that functions be packaged separately as COMDATs to exclude or order individual functions in a DLL or .exe file.

You can use the linker option /OPT (Optimizations) to exclude unreferenced packaged functions from the .exe file.

You can use the linker option /ORDER (Put Functions in Order) to include packaged functions in a specified order in the .exe file.

Inline functions are always packaged if they are instantiated as calls (which occurs, for example, if inlining is off or you take a function address). In addition, C++ member functions defined in the class declaration are automatically packaged; other functions are not, and selecting this option is required to compile them as packaged functions.

> **Note**
>
> The **/ZI** option, used for Edit and Continue, automatically sets the **/Gy** option.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.

3. Modify the **Enable Function-Level Linking** property.

To set this compiler option programmatically

- See EnableFunctionLevelLinking.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/GZ (Enable Stack Frame Run-Time Error Checking)

Article • 08/03/2021

Performs the same operations as the /RTC (Run-Time Error Checks) option. Deprecated.

Syntax

```
/GZ
```

Remarks

/GZ is only for use in a nonoptimized (/Od (Disable (Debug))) build.

/GZ is deprecated since Visual Studio 2005; use /RTC (Run-Time Error Checks) instead. For a list of deprecated compiler options, see **Deprecated and Removed Compiler Options** in Compiler Options Listed by Category.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/H (Restrict Length of External Names)

Article • 08/03/2021

Deprecated. Restricts the length of external names.

Syntax

> **/H***number*

Arguments

number
Specifies the maximum length of external names allowed in a program.

Remarks

By default, the length of external (public) names is 2,047 characters. This is true for C and C++ programs. Using **/H** can only decrease the maximum allowable length of identifiers, not increase it. A space between **/H** and *number* is optional.

If a program contains external names longer than *number*, the extra characters are ignored. If you compile a program without **/H** and if an identifier contains more than 2,047 characters, the compiler will generate Fatal Error C1064.

The limit on length includes any compiler-created leading underscore (_) or at sign (@). These characters are part of the identifier and take a significant location.

- The compiler adds a leading underscore (_) to names modified by the `__cdecl` (default) and `__stdcall` calling conventions, and a leading at sign (@) to names modified by the `__fastcall` calling convention.
- The compiler appends argument size information to names modified by the `__fastcall` and `__stdcall` calling conventions, and adds type information to C++ names.

You may find **/H** useful:

- When you create mixed-language or portable programs.
- When you use tools that impose limits on the length of external identifiers.

- When you want to restrict the amount of space that symbols use in a debug build.

The following example shows how using **/H** can actually introduce errors if identifier lengths are limited too much:

```
C++

// compiler_option_H.cpp
// compile with: /H5
// processor: x86
// LNK2005 expected
void func1(void);
void func2(void);

int main() { func1(); }

void func1(void) {}
void func2(void) {}
```

You must also be careful when using the **/H** option because of predefined compiler identifiers. If the maximum identifier length is too small, certain predefined identifiers will be unresolved as well as certain library function calls. For example, if the `printf` function is used and the option **/H5** is specified at compile time, the symbol **_prin** will be created in order to reference `printf`, and this will not be found in the library.

Use of **/H** is incompatible with /GL (Whole Program Optimization).

The **/H** option is deprecated since Visual Studio 2005; the maximum length limits have been increased and **/H** is no longer needed. For a list of deprecated compiler options, see **Deprecated and Removed Compiler Options** in Compiler Options Listed by Category.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/headerName` (Build a header unit from the specified header)

Article • 11/18/2022

Build the specified header file into a header unit (`.ifc` file).

Syntax

> **`/headerName:quote`** `header-filename`
>
> **`/headerName:angle`** `header-filename`

Arguments

`header-filename`
The name of a header file that the compiler should compile into a header unit (`.ifc` file).

Remarks

The **`/headerName:quote`** and **`/headerName:angle`** compiler options are available starting in Visual Studio 2019 version 16.10.

The **`/headerName`** compiler options, in all their forms, require the /std:c++20 or later compiler option (such as **`/std:c++latest`**).
If you specify a **`/headerName`** option, you must also specify /exportHeader.

- **`/headerName:quote`** looks up *`header-filename`* using the same rules as `#include "header-filename"` and builds it as a header unit (`.ifc` file).
- **`/headerName:angle`** looks up *`header-filename`* using the same rules as `#include <header-filename>` and builds it as a header unit (`.ifc` file).

For more information about the path searching rules for included files in quotes or angle brackets, see #include directive.

Examples

Given a project that references a header file it defines called *`m.h`*, the compiler option to compile it into a header unit looks similar to this example:

Bash

```
cl /std:c++latest /exportHeader /headerName:quote m.h /Fom.h.obj
```

The `/headerName:quote` and `/headerName:angle` options act like a flag and don't need an argument. The following examples are valid:

Bash

```
cl /std:c++latest /exportHeader /headerName:angle /MP /Fo.\ vector iostream
algorithm
cl /std:c++latest /exportHeader /headerName:quote /MP /Fo.\ my-utilities.h
a/b/my-core.h
```

You can specify multiple `/headerName` options on the same command line. Every argument after a `/headerName` option is processed with the specified include file lookup rules for quotes or angle brackets until the next `/headerName` option. The following example processes all the headers as the previous two command line examples in the same way as before. It looks up the headers using the lookup rules applied as if they had been specified as: `#include <vector>`, `#include <iostream>`, `#include <algorithm>`, `#include "my-utilties.h"`, and `#include "a/b/my-core.h"`:

Bash

```
cl /std:c++latest /exportHeader /headerName:angle /MP /Fo.\ vector iostream
algorithm /headerName:quote my-utilities.h a/b/my-core.h
```

To set this compiler option in the Visual Studio development environment

> **Note**
>
> You normally shouldn't set this option in the Visual Studio development environment. It's set by the build system.

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Set the **Configuration** drop-down to **All Configurations**. Set the **Platform** drop-down to **All Platforms**.

3. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

4. Modify the **Additional Options** property to add the `/headerName:quote` or `/headerName:angle` options and the header filenames the options apply to. Then, choose **OK** or **Apply** to save your changes.

See also

/exportHeader (Create header units)
/headerUnit (Use header unit IFC)
/reference (Use named module IFC)
/translateInclude (Translate include directives into import directives)

`/headerUnit` (Use header unit IFC)

Article • 05/28/2024

Imports a header unit. Tells the compiler where to find the *`.ifc`* file (the binary representation of the header unit) for the specified header.

Syntax

> **`/headerUnit`** *`header-filename`*`=`*`ifc-filename`*
> **`/headerUnit:quote`** *`header-filename`*`=`*`ifc-filename`*
> **`/headerUnit:angle`** *`header-filename`*`=`*`ifc-filename`*

Arguments

`header-filename`
During `import header-name;` the compiler resolves `header-name` to a file on disk. Use *`header-filename`* to specify that file. Once matched, the compiler opens the corresponding IFC named by *`ifc-filename`* for import.

`ifc-filename`
The name of a file that contains compiled header unit information. To import more than one header unit, add a separate **`/headerUnit`** option for each file.

Remarks

The **`/headerUnit`** compiler option requires /std:c++20 or later.

The **`/headerUnit`** compiler option is available in Visual Studio 2019 version 16.10 or later.

When the compiler comes across `import "file";` or `import <file>;` this compiler option helps the compiler find the compiled header unit (*`.ifc`*) for the specified header file. The path to this file can be expressed in these ways:

- **`/headerUnit`** looks up the compiled header unit in the current directory, or at the location specified by *`ifc-filename`*.
- **`/headerUnit:quote`** looks up the compiled header unit file using the same rules as `#include "file"`.

- `/headerUnit:angle` looks up the compiled header unit file using the same rules as `#include <file>`.

The compiler can't map a single *`header-name`* to multiple `.ifc` files. You can map multiple *`header-name`* arguments to a single `.ifc`. The contents of the `.ifc` are imported as if it was only the header specified by *`header-name`*.

The compiler implicitly enables the new preprocessor when this option is used. If any form of `/headerUnit` is specified on the command line, then /Zc:preprocessor is added to the command line by the compiler. To opt out of the implicit `/Zc:preprocessor`, specify: `/Zc:preprocessor-`

If you disable the new preprocessor, but a file you compile imports a header unit, the compiler will report an error.

Examples

Given a project that references two header files and their header units as listed in this table:

Expand table

Header file	IFC file
`C:\utils\util.h`	*`C:\util.h.ifc`*
`C:\app\app.h`	*`C:\app\app.h.ifc`*

The compiler options to reference the header units for these particular header files would look like this:

CMD

```
cl ... /std:c++latest /headerUnit C:\utils\util.h=C:\util.h.ifc
/headerUnit:quote app.h=app.h.ifc
```

To set this compiler option in the Visual Studio development environment

You normally shouldn't set this in the Visual Studio development environment. It's set by the build system.

See also

/exportHeader (Create header units)
/headerName (Create a header unit from the specified header)
/reference (Use named module IFC)
/translateInclude (Translate include directives into import directives)

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

/HELP (Compiler Command-Line Help)

Article • 08/03/2021

Displays a listing of compiler options to standard output.

Syntax

```
/HELP
/help
/?
```

Remarks

To set this compiler option in the Visual Studio development environment

- This compiler option should only be accessed from the command line.

To set this compiler option programmatically

- This compiler option cannot be changed programmatically.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/homeparams (Copy Register Parameters to Stack)

Article • 08/03/2021

Forces parameters passed in registers to also be written to their locations on the stack upon function entry.

Syntax

> **/homeparams**

Remarks

This compiler option is only available in the native and cross-compilers that target x64.

The x64 calling convention requires stack space to be allocated for all parameters, even for parameters passed in registers. For more information, see Parameter Passing. By default, the register parameters aren't copied into the stack space allocated for them in release builds. This makes it difficult to debug an optimized release build of your program.

For release builds, you can use the **/homeparams** option to force the compiler to copy register parameters to the stack, to ensure that you can debug your application. **/homeparams** does imply a performance disadvantage, because it requires an extra cycle to load the register parameters onto the stack.

In debug builds, the stack is always populated with parameters passed in registers.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/hotpatch (Create Hotpatchable Image)

Article • 12/09/2021

Prepares an image for hot patching.

Syntax

```
/hotpatch
```

Remarks

When **/hotpatch** is used in a compilation, the compiler ensures that the first instruction of each function is at least two bytes, and no jump within the function goes to the first instruction. These conditions are required for hot patching.

To complete the preparation for making an image hotpatchable, after you use **/hotpatch** to compile, you must use /FUNCTIONPADMIN (Create Hotpatchable Image) to link. When you compile and link an image by using one invocation of cl.exe, **/hotpatch** implies **/functionpadmin**.

Because instructions are always two bytes or larger on the ARM architecture, and because x64 compilation is always treated as if **/hotpatch** has been specified, you don't have to specify **/hotpatch** when you compile for these targets; however, you must still link by using **/functionpadmin** to create hotpatchable images for them. The **/hotpatch** compiler option only affects x86 compilation.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Add the compiler option to the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/I` (Additional include directories)

Article • 08/03/2021

Adds a directory to the list of directories searched for include files.

Syntax

> `/I` *directory*

Arguments

directory
The directory to add to the list of directories searched for include files. The space between `/I` and *directory* is optional. Directories that include spaces must be enclosed in double quotes. A directory may be an absolute path or a relative path.

Remarks

To add more than one directory, use this option more than once. Directories are searched only until the specified include file is found.

You can use this option on the same command line as the (/X (Ignore standard include paths)) option.

A #include directive can be specified in double-quote (or local-first) form, for example, `#include "local.h"`. Or, it can be specified in angle-bracket (or include-path-first) form, for example, `#include <iostream>`.

The compiler searches directories in the following order:

1. If the **`#include`** directive is specified using double-quote form, it first searches local directories. The search begins in the same directory as the file that contains the **`#include`** directive. If it fails to find the file, it searches next in the directories of the currently opened include files, in the reverse order in which they were opened. The search begins in the directory of the parent include file and continues upward through the directories of any grandparent include files.

2. If the **`#include`** directive is specified in angle-bracket form, or if the local directory search has failed, it searches directories specified by using the **`/I`** option, in the order they're specified on the command line.

3. Directories specified in the `INCLUDE` environment variable.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **General** property page.
3. Modify the **Additional Include Directories** property. You can specify more than one directory at a time in this property. Directories must be separated by a semicolon (`;`).

To set this compiler option programmatically

- See AdditionalIncludeDirectories.

Example

The following command looks for the include files requested by *`main.c`* in the following order: First, if specified by using double-quotes, local files are searched. Next, search continues in the *`\include`* directory, then in the *`\my\include`* directory, and finally in the directories assigned to the `INCLUDE` environment variable, in left to right order.

Windows Command Prompt

```
CL /I \include /I\my\include main.c
```

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/ifcOutput

Article • 11/22/2022

This switch tells the compiler where to output built *.ifc* files. If the destination is a directory, then the compiler generates the name of each *.ifc* file based on the interface name or the header unit name.

Syntax

> **/ifcOutput** *filename*
> **/ifcOutput** *directory*

Remarks

By default, the compiler derives the name for each generated *.ifc* file from the module interface name. For example, given a module name `MyModule`, the generated *.ifc* will be named *MyModule.ifc*, unless you override the name with the `/ifcOutput` switch.

Use this switch to specify an alternative *.ifc* filename or directory. If you want to use the default built *.ifc* filenames, but specify a directory where they should be built, ensure that you add a trailing backslash (`\`) to the directory name.

When you're building multiple *.ifc* files, only use the directory form of the `/ifcOutput` switch. If you provide multiple `/ifcOutput` switches, the compiler only uses the last one.

If you build with the /MP (Build with multiple processes) switch, we recommend that you use the directory form of the `/ifcOutput` switch if you have multiple input module files.

In the following example, the *.ifc* file for module `m` defined in *m.ixx* is built as `c:\example\m.ifc`.

Bash

```
cl ... /c /std:c++latest m.ixx /ifcOutput c:\example\
```

In the following example, the built *.ifc* file for module `m` defined in *m.ixx** is built as `c:\example\MyModule.ifc`:

Bash

```
cl ... /c /std:c++latest m.ixx /ifcOutput c:\example\MyModule.ifc
```

To set this compiler option in the Visual Studio development environment

1. To apply the `/ifcOutput` option to one file in the IDE, select the file in **Solution Explorer**. Right-click to open the context menu and select **Properties** to open the **Property Pages** dialog.

2. Set the **Configuration** dropdown to **All Configurations**. Set the **Platform** dropdown to **All Platforms**.

3. Open the **Configuration Properties** > **C/C++** > **Output Files** property page.

4. Use the dropdown control to modify the **Module Output File Name** property to a directory name (ending in `\`) or an alternate file name. Or you can specify a directory + file name, for example, `c:\example\mymodule.ifc`. Choose **OK** or **Apply** to save your changes.

Alternatively, you can specify the `/ifcOutput` switch with a right-click on the project name in the **Solution Explorer** > **Configuration Properties** > **C/C++** > **Command Line**.

See also

Overview of modules in C++
Using C++ Modules in MSVC from the Command Line

`/ifcMap`

Article • 10/17/2023

This switch tells the compiler where to find the IFC reference map file, which maps references to named modules and header units to their corresponding IFC (`.ifc`) files.

Syntax

> **`/ifcMap`** *`filename`*

Remarks

The `*filename*` argument specifies the IFC reference map file. It can be relative to the compiler's working directory, or an absolute path.

You can provide multiple `/ifcMap` arguments to the compiler.

The IFC reference map file format is a subset of the TOML file format. The IFC reference map file can contain a mix of `[[module]]` and `[[header-unit]]` references.

Syntax errors or unrecognized table names result in compiler error `C7696` (TOML parse error).

Map named modules

The format of the IFC reference map file for named modules is:

```
# Using literal strings
[[module]]
name = 'M'
ifc = 'C:\modules\M.ifc'

# Using basic strings
[[module]]
name = "N"
ifc = "C:\\modules\\N.ifc"
```

This IFC reference map file maps the named modules `'M'` and `'N'` to their respective IFC files. The equivalent `/reference' is:

Windows Command Prompt

```
/reference M=C:\modules\M.ifc /reference N=C:\modules\N.ifc
```

For more information about what types of module names are valid for the `name` field, see /reference remarks.

Map header units

The format of the IFC reference map file for header units is:

```
# Using literal strings
[[header-unit]]
name = ['quote', 'my-utility.h']
ifc = 'C:\header-units\my-utility.h.ifc'

[[header-unit]]
name = ['angle', 'vector']
ifc = 'C:\header-units\vector.ifc'

# Using basic strings
[[header-unit]]
name = ["quote", "my-engine.h"]
ifc = "C:\\header-units\\my-engine.h.ifc"

[[header-unit]]
name = ["angle", "algorithm"]
ifc = "C:\\header-units\\algorithm.ifc"
```

This IFC reference map file maps `"my-utility.h"` to `C:\header-units\my-utility.h.ifc`, and `<vector>` to `C:\header-units\vector.ifc`, and so on. The equivalent /headerUnit is:

Windows Command Prompt

```
/headerUnit:quote my-utility=C:\header-units\my-utility.h.ifc
/headerUnit:angle vector=C:\header-units\vector.ifc /headerUnit:quote my-
engine.h=C:\header-units\my-engine.h.ifc /headerUnit:angle
algorithm=C:\header-units\algorithm.ifc
```

When `[[header-unit]]` is specified in an IFC reference map file, the compiler implicitly enables /Zc:preprocessor, just as it's implicitly enabled when /headerUnit is used. For more information about the behavior of the `angle` and `quote` lookup methods, see /headerUnit remarks.

See also

Overview of modules in C++
Walkthrough: Build and import header units in Visual C++ projects
Using C++ Modules in MSVC from the Command Line

`/interface`

Article • 11/18/2022

This switch instructs the compiler to treat the input file on the command line as a module interface unit.

Syntax

> **`/interface`** *`filename`*

Remarks

Use this switch when a module interface has a different extension than `.ixx`.

In the following example, the module interface has a `.cppm` extension instead of `.ixx`, so the `/interface` switch is used to compile it as a module interface:

```
Bash

cl /c /std:c++latest /interface /TP my-module.cppm
```

The compiler derives the name for the generated `.ifc` file from the module interface name. For example, given a module name `MyModule` defined in `my-module.cppm`, the generated `.ifc` will be named `MyModule.ifc`.

This switch must be used in with the /TP (Specify source file type) compiler flag.

`/interface` is available in Visual Studio 2019 version 16.10, or later.
`/interface` requires /std:c++20 or later.

To set this compiler option in the Visual Studio development environment

You normally shouldn't set this option in the Visual Studio development environment unless you use a different extension for your module interface files. By default, the build system applies this option to files that have a *`.ixx`** extension.

1. To apply the **`/interface`** option to a file explicitly in the IDE, select the file in **Solution Explorer**. Right-click to open the context menu and select **Properties** to

open the Property Pages dialog.

2. Set the **Configuration** dropdown to **All Configurations**. Set the **Platform** dropdown to **All Platforms**.

3. Open the **Configuration Properties** > **C/C++** > **Advanced** property page.

4. Use the dropdown control to modify the **Compile As** property to **Compile as C++ Module Code (/interface)**. Choose **OK** or **Apply** to save your changes.

See also

Overview of modules in C++
Using C++ Modules in MSVC from the Command Line

`/internalPartition`

Article • 11/18/2022

Use the `/internalPartition` compiler option to treat the input file as an *internal partition unit*, which is a module partition implementation unit that doesn't contribute to the external interface of the module.

Syntax

> `/internalPartition` *`filename`*

Remarks

The following example demonstrates how to use the `/internalPartition` option:

Source file `m-internals.cpp`:

C++

```
// m-internals.cpp
module m:internals;

void internalFunc() {} // cannot have `export` since this is an internal
partition
```

Source file `m.ixx`:

C++

```
// m.ixx
export module m;
import :internals; // Cannot export this partition.

export
void wrapper() { internalFunc(); }
```

To compile this interface:

Bash

```
cl /std:c++latest /internalPartition /c m-internals.cpp
```

This option can't be used with the /interface compiler option.

`/internalPartition` is available in Visual Studio 2019 version 16.10, or later.
`/internalPartition` requires /std:c++20 or later.

To set this compiler option in the Visual Studio development environment

You normally shouldn't set this option in the Visual Studio development environment unless you use a different extension for your partition files. By default, the build system applies this option to files that have a `.ixx*` extension.

1. To apply the `/internalPartition` option to a file explicitly in the IDE, select the file in **Solution Explorer**. Right-click to open the context menu and select **Properties** to open the Property Pages dialog.

2. Set the **Configuration** dropdown to **All Configurations**. Set the **Platform** dropdown to **All Platforms**.

3. Open the **Configuration Properties** > **C/C++** > **Advanced** property page.

4. Use the dropdown control to modify the **Compile As** property to **Compile as C++ Module Internal Partition (/internalPartition)**. Choose **OK** or **Apply** to save your changes.

See also

Overview of modules in C++
Using C++ Modules in MSVC from the Command Line
C++ Modules conformance improvements with MSVC in Visual Studio 2019 16.5

/J (Default char Type Is unsigned)

Article • 08/03/2021

Changes the default `char` type from `signed char` to `unsigned char`, and the `char` type is zero-extended when it is widened to an `int` type.

Syntax

```
/J
```

Remarks

If a `char` value is explicitly declared as `signed`, the **/J** option does not affect it, and the value is sign-extended when it is widened to an `int` type.

The **/J** option defines `_CHAR_UNSIGNED`, which is used with `#ifndef` in the LIMITS.h file to define the range of the default `char` type.

ANSI C and C++ do not require a specific implementation of the `char` type. This option is useful when you are working with character data that will eventually be translated into a language other than English.

> **Note**
>
> If you use this compiler option with ATL/MFC, an error might be generated. Although you could disable this error by defining `_ATL_ALLOW_CHAR_UNSIGNED`, this workaround is not supported and may not always work.

To set this compiler option in the Visual Studio development environment

1. Open your project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

3. In **Additional Options**, enter the **/J** compiler option.

To set this compiler option programmatically

- See DefaultCharIsUnsigned.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Set C++ compiler and build properties in Visual Studio

`/JMC` (Just My Code debugging)

Article • 03/02/2022

Specifies compiler support for native *Just My Code* debugging in the Visual Studio debugger. This option supports the user settings that allow Visual Studio to step over system, framework, library, and other non-user calls, and to collapse those calls in the call stack window. The `/JMC` compiler option is available starting in Visual Studio 2017 version 15.8.

Syntax

> `/JMC`[`-`]

Remarks

The Visual Studio Just My Code settings specify whether the Visual Studio debugger steps over system, framework, library, and other non-user calls. The `/JMC` compiler option enables support for Just My Code debugging in your native C++ code. When `/JMC` is enabled, the compiler inserts calls to a helper function, `__CheckForDebuggerJustMyCode`, in the function prolog. The helper function provides hooks that support Visual Studio debugger Just My Code step operations. To enable Just My Code in the Visual Studio debugger, on the menu bar, choose **Tools** > **Options**, and then set the option in **Debugging** > **General** > **Enable Just My Code**.

The `/JMC` option requires that your code links to the C Runtime Library (CRT), which provides the `__CheckForDebuggerJustMyCode` helper function. If your project doesn't link to the CRT, you may see linker error **LNK2019: unresolved external symbol __CheckForDebuggerJustMyCode**. To resolve this error, either link to the CRT, or disable the `/JMC` option.

When the `/JMC` option is enabled, the PDB file is annotated with extra line number information. In versions before Visual Studio 2019 version 16.8, this information may appear in code coverage reports as occurring at line 15732480 (0xF00F00) or 16707566 (0xFEEFEE). These fictitious line numbers are used as markers to delineate user code from non-user code. To include non-user code in code coverage reports without these unexpected line numbers, build your code with the `/JMC-` option.

By default, the `/JMC` compiler option is off. However, starting in Visual Studio 2017 version 15.8 this option is enabled in most Visual Studio project templates. To explicitly

disable this option, use the `/JMC-` option on the command line. In Visual Studio, open the project Property Pages dialog box, and change the **Support Just My Code Debugging** property in the **Configuration Properties** > **C/C++** > **General** property page to **No**.

For more information, see C++ Just My Code in Specify whether to debug only user code using Just My Code in Visual Studio, and the Visual C++ Team Blog post Announcing C++ Just My Code Stepping in Visual Studio.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **General** property page.
3. Modify the **Support Just My Code Debugging** property.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax

/jumptablerdata (put switch case jump tables in `.rdata`)

Article • 07/11/2023

Puts the generated switch case jump tables in the `.rdata` section instead of alongside code in the `.text` section.

Syntax

```
C++

/jumptablerdata
```

Remarks

Putting jump tables generated for switch case statements in the `.rdata` section prevents the jump table from being loaded into both the instruction cache (iCache) and data cache (dCache), potentially increasing performance. The `.rdata` section is where const initialized data is stored.

> **ⓘ Important**
>
> This flag only applies to x64 code. This flag was introduced in Visual Studio 17.7.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/jumptablerdata` and then choose **OK**.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/kernel (Create kernel mode binary)

Article • 08/03/2021

Creates a binary that can be executed in the Windows kernel. The code in the current project gets compiled and linked by using a simplified set of C++ language features that are specific to code that runs in kernel mode.

Syntax

> `/kernel`

Remarks

Specifying the `/kernel` option tells the compiler and linker to arbitrate which language features are permissible in kernel mode and to make sure that you have sufficient expressive power to avoid runtime instability that is unique to kernel mode C++. It's done by prohibiting the use of C++ language features that are disruptive in kernel mode. The compiler produces warnings for C++ language features that are potentially disruptive but can't be disabled.

The `/kernel` option applies to both the compiler and linker phases of a build and is set at the project level. Pass the `/kernel` switch to indicate to the compiler that the resulting binary, after linking, should be loaded into the Windows kernel. The compiler will narrow the spectrum of C++ language features to a subset that is compatible with the kernel.

The following table lists changes in compiler behavior when `/kernel` is specified.

Behavior type	`/kernel` **behavior**
C++ exception handling	Disabled. All instances of the `throw` and `try` keywords emit a compiler error (except for the exception specification `throw()`). No `/EH` options are compatible with `/kernel`, except for `/EH-`.
RTTI	Disabled. All instances of the `dynamic_cast` and `typeid` keywords emit a compiler error, unless `dynamic_cast` is used statically.
`new` and `delete`	You must explicitly define the `new()` or `delete()` operator. The compiler and runtime don't supply a default definition.

Custom calling conventions, the /GS build option, and all optimizations are permitted when you use the `/kernel` option. Inlining is largely not affected by `/kernel`, with the same semantics honored by the compiler. If you want to make sure that the `__forceinline` inlining qualifier is honored, you must make sure that warning C4714 is enabled so that you know when a particular `__forceinline` function isn't inlined.

There's no `#pragma` equivalent to control this option.

When the compiler is passed the `/kernel` switch, it predefines a preprocessor macro that's named `_KERNEL_MODE` and has the value **1**. You can use this macro to conditionally compile code based on whether the execution environment is in user mode or kernel mode. For example, the following code specifies that the `MyNonPagedClass` class should be in a non-pageable memory segment when it's compiled for kernel mode execution.

C++

```
#ifdef _KERNEL_MODE
#define NONPAGESECTION __declspec(code_seg("$kerneltext$"))
#else
#define NONPAGESECTION
#endif

class NONPAGESECTION MyNonPagedClass
{
   // ...
};
```

Some of the following combinations of target architecture and the `/arch` option produce an error when they're used with `/kernel`:

- `/arch:SSE`, `/arch:SSE2`, `/arch:AVX`, `/arch:AVX2`, and `/arch:AVX512` aren't supported on x86. Only `/arch:IA32` is supported with `/kernel` on x86.
- `/arch:AVX`, `/arch:AVX2`, and `/arch:AVX512` aren't supported with `/kernel` on x64.

Building with `/kernel` also passes `/kernel` to the linker. Here's how the option affects linker behavior:

- Incremental linking is disabled. If you add `/incremental` to the command line, the linker emits this fatal error:

 fatal error LNK1295: '/INCREMENTAL' not compatible with '/KERNEL' specification; link without '/INCREMENTAL'

- The linker inspects each object file (or any included archive member from static libraries) to see whether it could have been compiled by using the `/kernel` option but wasn't. If any instances meet this criterion, the linker still successfully links but might issue a warning, as shown in the following table.

Command	`/kernel` obj	non-`/kernel` obj, MASM obj, or cvtres obj	Mix of `/kernel` and non-`/kernel` objs
`link /kernel`	Yes	Yes	Yes with warning LNK4257
`link`	Yes	Yes	Yes

 LNK4257 linking object not compiled with /KERNEL; image may not run

The `/kernel` option and the `/driver` option operate independently. They have no effect on each other.

To set the /kernel compiler option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In the **Additional options** box, add `/kernel`. Choose **OK** or **Apply** to save your changes.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/link (Pass Options to Linker)

Article • 08/03/2021

Passes one or more linker options to the linker.

Syntax

> **/link** *linker-options*

Arguments

linker-options
The linker option or options to be passed to the linker.

Remarks

The **/link** option and its linker options must appear after any file names and CL options. A space is required between **/link** and any linker options. For more information, see MSVC linker reference.

Example

This sample command line compiles *hello.cpp* and links it to the existing object file *there.obj*. It then passes an additional **/VERSION** command to the linker:

```
cl /W4 /EHsc hello.cpp there.obj /link /VERSION:3.14
```

To set this compiler option in the Visual Studio development environment

The IDE normally sends separate commands to compile and link your code. You can set linker options in your project property pages.

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Linker** folder.

3. Modify one or more properties. Choose **OK** to save your changes.

To set this compiler option programmatically

- This compiler option can't be changed programmatically.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/LN (Create MSIL Module)

Article • 08/03/2021

Specifies that an assembly manifest should not be inserted into the output file.

Syntax

```
/LN
```

Remarks

By default, **/LN** is not in effect (an assembly manifest is inserted into the output file).

When **/LN** is used, one of the /clr (Common Language Runtime Compilation) options must also be used.

A managed program that does not have an assembly metadata in the manifest is called a module. If you compile with /c (Compile Without Linking) and **/LN**, specify /NOASSEMBLY (Create a MSIL Module) in the linker phase to create the output file.

You may want to create modules if you want to take a component-based approach to building assemblies. That is, you can author types and compile them into modules. Then, you can generate an assembly from one or more modules. For more information on creating assemblies from modules, see .netmodule Files as Linker Input or Al.exe (Assembly Linker).

The default file extension for a module is .netmodule.

In releases before Visual Studio 2005, a module was created with **/clr:noAssembly**.

The MSVC linker accepts .netmodule files as input and the output file produced by the linker will be an assembly or .netmodule with no run-time dependence on any of the .netmodules that were input to the linker. For more information, see .netmodule Files as Linker Input.

To set this compiler option in the Visual Studio development environment

- Specify /NOASSEMBLY (Create a MSIL Module) in the linker phase to create the output file.

To set this compiler option programmatically

- This compiler option cannot be changed programmatically.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/MD, /MT, /LD (Use runtime library)

Article • 01/14/2025

Indicates whether a multithreaded module is a DLL and specifies retail or debug versions of the runtime library.

Syntax

```
/MD[d]
/MT[d]
/LD[d]
```

Remarks

Expand table

Option	Description
/MD	Use the multithread-specific and DLL-specific version of the runtime library. Defines `_MT` and `_DLL`. The linker uses the `MSVCRT.lib` import library to resolve runtime symbols.
/MDd	Use the debug multithread-specific and DLL-specific version of the runtime library. Defines `_DEBUG`, `_MT`, and `_DLL`. The linker uses the `MSVCRTD.lib` import library to resolve runtime symbols.
/MT	Use the multithread, static version of the runtime library. Defines `_MT`. The linker uses `LIBCMT.lib` to resolve runtime symbols.
/MTd	Use the debug multithread, static version of the runtime library. Defines `_DEBUG` and `_MT`. The linker uses `LIBCMTD.lib` to resolve runtime symbols.
/LD	Create a DLL. Passes the **/DLL** option to the linker. The linker looks for, but does not require, a `DllMain` function. If you don't write a `DllMain` function, the linker inserts a `DllMain` function that returns TRUE. Links the DLL startup code. Creates an import library (`.lib`), if an export (`.exp`) file is not specified on the command line. You link the import library to applications that call your DLL.

Option	Description
	Interprets /Fe (Name EXE File) as naming a DLL rather than an `.exe` file. By default, the program name becomes *basename*.dll instead of *basename*.exe. Implies **/MT** unless you explicitly specify **/MD**.
/LDd	Create a debug DLL. Defines `_MT` and `_DEBUG`.

For more information about C runtime libraries and which libraries are used when you compile with /clr (Common Language Runtime Compilation), see CRT Library Features.

All modules passed to a given invocation of the linker must have been compiled with the same runtime library compiler option (**/MD**, **/MT**, **/LD**).

For more information about how to use the debug versions of the runtime libraries, see C runtime Library Reference.

For more about DLLs, see Create C/C++ DLLs in Visual Studio.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify the **Runtime Library** property.

To set this compiler option programmatically

- See RuntimeLibrary.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
The Great C Runtime (CRT) Refactoring

Feedback

Was this page helpful?

Yes No

Provide product feedback | Get help at Microsoft Q&A

`/exportHeader` (Create header units)

Article • 11/18/2022

Tells the compiler to create the header units specified by the input arguments. The compiler generates header units as IFC (*`.ifc`*) files.

Syntax

> **`/exportHeader /headerName:angle`** *`header-name`*
>
> **`/exportHeader /headerName:quote`** *`header-name`*
>
> **`/exportHeader`** *`full path to header file`*

Arguments

The argument to `/exportHeader` is a `/headerName` command-line option that specifies the name, *`header-name`*, of the header file to export.

Remarks

`/exportHeader` is available starting in Visual Studio 2019 version 16.10.

The **`/exportHeader`** compiler option requires you enable the /std:c++20 or later compiler option (such as **`/std:c++latest`**).

One **`/exportHeader`** compiler option can specify as many header-name arguments as your build requires. You don't need to specify them separately.

The compiler implicitly enables the new preprocessor when this option is used. That is, /Zc:preprocessor is added to the command line by the compiler if any form of `/exportHeader` is used on the command line. To opt out of the implicit `/Zc:preprocessor`, use: `/Zc:preprocessor-`

By default, the compiler doesn't produce an object file when a header unit is compiled. To produce an object file, specify the **`/Fo`** compiler option. For more information, see /Fo (Object File Name).

You may find it helpful to use the complementary option **`/showResolvedHeader`**. The **`/showResolvedHeader`** option prints an absolute path to the file the *`header-name`* argument resolves to.

`/exportHeader` can handle multiple inputs at once, even under `/MP`. We recommended you use `/ifcOutput <directory>` to create a separate `.ifc` file for each compilation.

Examples

To build a header unit such as `<vector>` might look like:

Windows Command Prompt

```
cl . . . /std:c++latest /exportHeader /headerName:angle vector
```

Building a local project header such as `"utils/util.h"` might look like:

Windows Command Prompt

```
cl . . . /std:c++latest /exportHeader /headerName:quote util/util.h
```

To set this compiler option in the Visual Studio development environment

You normally shouldn't set this option in the Visual Studio development environment unless you use a different extension for your header files. By default, the build system applies this option to compiled files that have a `.h` extension, or no extension.

1. To apply the `/exportHeader` option to a file explicitly in the IDE, select the file in **Solution Explorer**. Right-click to open the context menu and select **Properties** to open the Property Pages dialog.

2. Set the **Configuration** dropdown to **All Configurations**. Set the **Platform** dropdown to **All Platforms**.

3. Open the **Configuration Properties** > **C/C++** > **Advanced** property page.

4. Use the dropdown control to modify the **Compile As** property to **Compile as C++ Header Unit (/exportHeader)**. Choose **OK** or **Apply** to save your changes.

See also

/headerName (Build a header unit from the specified header)
/headerUnit (Use header unit IFC)

/reference (Use named module IFC)

/translateInclude (Translate include directives into import directives)

`/reference` (Use named module IFC)

Article • 02/18/2022

Tells the compiler to use an existing IFC (*`.ifc`*) for the current compilation.

Syntax

> **`/reference`** *`module-name=filename`*
>
> **`/reference`** *`filename`*

Arguments

`filename`
The name of a file that contains *IFC data*, which is prebuilt module information. To import more than one module, include a separate **`/reference`** option for each file.

`module-name`
A valid name of an exported primary module interface unit name or full module partition name.

Remarks

In most cases, you won't need to specify this switch because the project system discovers module dependencies within a solution automatically.

The **`/reference`** compiler option requires you enable the /std:c++20 or later compiler option (such as **`/std:c++latest`**). The **`/reference`** option is available starting in Visual Studio 2019 version 16.10.

If the **`/reference`** argument is a *`filename`* without a *`module-name`*, the file gets opened at runtime to verify the *`filename`* argument names a specific import. It can result in slower runtime performance in scenarios that have many **`/reference`** arguments.

The *`module-name`* must be a valid primary module interface unit name or full module partition name. Examples of primary module interface names include:

- `M`
- `M.N.O`
- `MyModule`

- `my_module`

Examples of full module partition names include:

- `M:P`
- `M.N.O:P.Q`
- `MyModule:Algorithms`
- `my_module:algorithms`

If a module reference is created using a *`module-name`*, other modules on the command line don't get searched if the compiler encounters an import of that name. For example, given this command line:

Windows Command Prompt

```
cl ... /std:c++latest /reference m.ifc /reference m=n.ifc
```

In the case above, if the compiler sees `import m;` then *`m.ifc`* doesn't get searched.

Examples

Given three modules as listed in this table:

Module	IFC file
`M`	*`m.ifc`*
`M:Part1`	*`m-part1.ifc`*
`Core.Networking`	*`Networking.ifc`*

The reference options using a *`filename`* argument would be like this:

Windows Command Prompt

```
cl ... /std:c++latest /reference m.ifc /reference m-part.ifc /reference
Networking.ifc
```

The reference options using *`module-name=filename`* would be like this:

Windows Command Prompt

```
cl ... /std:c++latest /reference m=m.ifc /reference M:Part1=m-part.ifc
/reference Core.Networking=Networking.ifc
```

See also

/scanDependencies (List module dependencies in standard form)
/sourceDependencies:directives (List module and header unit dependencies)
/headerUnit (Use header unit IFC)
/exportHeader (Create header units)

`/MP` (Build with multiple processes)

Article • 06/16/2022

The `/MP` option can reduce the total time to compile the source files on the command line. The `/MP` option causes the compiler to create one or more copies of itself, each in a separate process. Then these instances simultaneously compile the source files. In some cases, the total time to build the source files can be reduced significantly.

Syntax

> `/MP`[*`processMax`*]

Arguments

`processMax`
(Optional) The maximum number of processes that the compiler can create.

The *`processMax`* argument must range from 1 through 65536. Otherwise, the compiler issues warning message **D9014**, ignores the *`processMax`* argument, and assumes the maximum number of processes is 1.

If you omit the *`processMax`* argument, the compiler retrieves the number of effective processors on your computer from the operating system, and creates a process for each processor.

Remarks

The `/MP` compiler option can significantly reduce build time when you compile many files. To improve build time, the compiler creates up to *`processMax`* copies of itself, and then uses those copies to compile your source files at the same time. The `/MP` option applies to compilations, but not to linking or link-time code generation. By default the `/MP` option is off.

The improvement in build time depends on the number of processors on a computer, the number of files to compile, and the availability of system resources, such as I/O capacity. Experiment with the `/MP` option to determine the best setting to build a particular project. For advice to help you make that decision, see Guidelines.

Incompatible options and language features

The `/MP` option is incompatible with some compiler options and language features. If you use an incompatible compiler option with the `/MP` option, the compiler issues warning **D9030** and ignores the `/MP` option. If you use an incompatible language feature, the compiler issues error C2813 then ends or continues depending on the current compiler warning level option.

> **Note**
>
> Most options are incompatible because if they were permitted, the concurrently executing compilers would write their output at the same time to the console or to a particular file. As a result, the output would intermix and be garbled. In some cases, the combination of options would make the performance worse.

The following table lists compiler options and language features that are incompatible with the `/MP` option:

Option or Language Feature	Description
#import preprocessor directive	Converts the types in a type library into C++ classes, and then writes those classes to a header file.
/E, /EP	Copies preprocessor output to the standard output (`stdout`).
/Gm	Deprecated. Enables an incremental rebuild.
/showIncludes	Writes a list of include files to the standard error (`stderr`).
/Yc	Writes a precompiled header file.

Diagnostic messages

If you specify an option or language feature that is incompatible with the `/MP` option, you'll receive a diagnostic message. The following table lists the messages and the behavior of the compiler:

Diagnostic Message	Description	Compiler Behavior
C2813	The `#import` directive isn't compatible with the `/MP` option.	The compilation ends unless a compiler warning level option specifies otherwise.

Diagnostic Message	Description	Compiler Behavior
D9014	An invalid value is specified for the *`processMax`* argument.	The compiler ignores the invalid value and assumes a value of 1.
D9030	The specified option is incompatible with `/MP`.	The compiler ignores the `/MP` option.

Guidelines

Measure performance

Use total build time to measure performance. You can measure the build time with a physical clock, or you can use software that calculates the difference between when the build starts and stops. If your computer has multiple processors, a physical clock might yield more accurate results than a software time measurement.

Effective processors

A computer can have one or more virtual processors, which are also known as *effective processors*, for each of its physical processors. Each physical processor can have one or more cores, and if the operating system enables hyperthreading for a core, each core appears to be two virtual processors.

For example, a computer has one effective processor if it has one physical processor that has one core, and hyperthreading is disabled. In contrast, a computer has eight effective processors if it has two physical processors, each of which has two cores, and all the cores have hyperthreading enabled. That is, (8 effective processors) = (2 physical processors) x (2 cores per physical processor) x (2 effective processors per core because of hyperthreading).

If you omit the *`processMax`* argument in the `/MP` option, the compiler obtains the number of effective processors from the operating system, and then creates one process per effective processor. However, the compiler can't guarantee which process executes on a particular processor; the operating system makes that decision.

Number of processes

The compiler calculates the number of processes that it will use to compile the source files. That value is the lesser of the number of source files that you specify on the

command line, and the number of processes that you explicitly or implicitly specify with the `/MP` option. You can explicitly set the maximum number of processes if you provide the *`processMax`* argument of the `/MP` option. Or you can use the default, which is equal to the number of effective processors in a computer, if you omit the *`processMax`* argument.

For example, suppose you specify the following command line:

```
cl /MP7 a.cpp b.cpp c.cpp d.cpp e.cpp
```

In this case, the compiler uses five processes because that is the lesser of five source files and a maximum of seven processes. Alternatively, suppose your computer has two effective processors and you specify the following command line:

```
cl /MP a.cpp b.cpp c.cpp
```

In this case, the operating system reports two processors, so the compiler uses two processes in its calculation. As a result, the compiler uses two processes to execute the build because that's the lesser of two processes and three source files.

Source files and build order

The source files might not be compiled in the same order in which they appear on the command line. Although the compiler creates a set of processes that contain copies of the compiler, the operating system schedules when each process executes. The `/MP` option can't guarantee that the source files will be compiled in a particular order.

A source file is compiled when a process is available to compile it. If there are more files than processes, the first set of files is compiled by the available processes. The remaining files are processed when a process finishes handling a previous file and is available to work on one of the remaining files.

Don't specify the same source file multiple times on a command line. Multiple specifications could occur, for example, if a tool automatically creates a makefile that's based on dependency information in a project. If you don't specify the `/MP` option, the compiler processes the list of files sequentially and recompiles each occurrence of the file. However, if you specify the `/MP` option, different compiler instances might compile the same file at the same time. The different instances may try to write to the same output file at the same time. One compiler instance acquires exclusive write access to the output file and succeed, and the other compiler instances fail with a file access error.

Using type libraries (`#import`)

The compiler doesn't support the use of the #import directive with the `/MP` switch. If possible, follow these steps to work around this problem:

- Move all the `#import` directives in your various source files to one or more files, and then compile those files without the `/MP` option. The result is a set of generated header files.
- In your remaining source files, insert #include directives that specify the generated headers, and then compile your remaining source files by using the `/MP` option.

Visual Studio Project settings

The MSBuild tool

Visual Studio uses the MSBuild tool (`msbuild.exe`) to build solutions and projects. The `/maxcpucount:number` (or `/m:number`) command-line option of the MSBuild tool can build multiple projects at the same time. And the `/MP` compiler option can build multiple compilation units at the same time. If it's appropriate for your application, improve your solution's build time by using either or both `/MP` and `/maxcpucount`.

The build time of your solution partly depends on the number of processes that perform the build. The *`number`* argument of the /maxcpucount MSBuild option specifies the maximum number of projects to build at the same time. Similarly, the *`processMax`* argument of the `/MP` compiler option specifies the maximum number of compilation units to build at the same time. If the `/maxcpucount` option specifies *P* projects and the `/MP` option specifies *C* processes, a maximum of *P* x *C* processes execute at the same time.

The guideline for deciding whether to use MSBuild or `/MP` technology is as follows:

- If there are many projects with few files in each project, use the MSBuild tool with the `/maxcpucount` option.
- If there are few projects with many files in each project, use the `/MP` option.
- If the number of projects and files per project is balanced, use both MSBuild and `/MP`. Initially set the `/maxcpucount` option to the number of projects to build and the `/MP` option to the number of processors on your computer. Measure performance and then adjust your settings to yield the best results. Repeat that cycle until you're satisfied with the total build time.

See also

#import directive

MSBuild command-line reference

/Zf (Faster PDB generation)

/nologo (Suppress Startup Banner) (C/C++)

Article • 08/03/2021

Suppresses the display of the copyright banner when the compiler starts up and display of informational messages during compiling.

Syntax

```
/nologo
```

Remarks

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **General** property page.
3. Modify the **Suppress Startup Banner** property.

To set this compiler option programmatically

- See SuppressStartupBanner.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/O` options (Optimize code)

Article • 08/03/2021

The `/O` options control various optimizations that help you create code for maximum speed or minimum size.

- /O1 sets a combination of optimizations that generate minimum size code.
- /O2 sets a combination of optimizations that optimizes code for maximum speed.
- /Ob controls inline function expansion.
- /Od disables optimization, to speed compilation and simplify debugging.
- /Og (deprecated) enables global optimizations.
- /Oi generates intrinsic functions for appropriate function calls.
- /Os tells the compiler to favor optimizations for size over optimizations for speed.
- /Ot (a default setting) tells the compiler to favor optimizations for speed over optimizations for size.
- /Ox is a combination option that selects several of the optimizations with an emphasis on speed. `/Ox` is a strict subset of the `/O2` optimizations.
- /Oy suppresses the creation of frame pointers on the call stack for quicker function calls.

Remarks

You can combine multiple `/O` options into a single option statement. For example, `/Odi` is the same as `/Od /Oi`. Certain options are mutually exclusive and cause a compiler error if used together. For more information, see the individual `/O` options.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/O1`, `/O2` (Minimize Size, Maximize Speed)

Article • 08/03/2021

Selects a predefined set of options that affect the size and speed of generated code.

Syntax

> `/O1`
>
> `/O2`

Remarks

The `/O1` and `/O2` compiler options are a quick way to set several specific optimization options at once. The `/O1` option sets the individual optimization options that create the smallest code in the majority of cases. The `/O2` option sets the options that create the fastest code in the majority of cases. The `/O2` option is the default for release builds. This table shows the specific options that are set by `/O1` and `/O2`:

Option	Equivalent to
`/O1` (Minimize Size)	/Og /Os /Oy /Ob2 /GF /Gy
`/O2` (Maximize Speed)	/Og /Oi /Ot /Oy /Ob2 /GF /Gy

`/O1` and `/O2` are mutually exclusive.

> ⓘ Note
>
> **x86-specific**
> These options imply the use of the Frame-Pointer Omission (**/Oy**) option.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Optimization** property page.

3. Modify the **Optimization** property.

To set this compiler option programmatically

- See Optimization.

See also

/O options (Optimize code)
MSVC compiler options
MSVC compiler command-line syntax
/EH (Exception handling model)

/Ob (Inline Function Expansion)

Article • 08/03/2021

Controls inline expansion of functions. By default, when optimizing, expansion occurs at the compiler's discretion on all functions, often referred to as *auto-inlining*.

Syntax

> **/Ob{0|1|2|3}**

Arguments

0
The default value under /Od. Disables inline expansions.

1
Allows expansion only of functions marked inline, __inline, or __forceinline, or in a C++ member function defined in a class declaration.

2
The default value under /O1 and /O2. Allows the compiler to expand any function not explicitly marked for no inlining.

3
This option specifies more aggressive inlining than **/Ob2**, but has the same restrictions. The **/Ob3** option is available starting in Visual Studio 2019.

Remarks

The compiler treats the inline expansion options and keywords as suggestions. There's no guarantee that any function will be expanded inline. You can disable inline expansions, but you can't force the compiler to inline a particular function, even when using the `__forceinline` keyword.

To exclude functions from consideration as candidates for inline expansion, you can use __declspec(noinline), or a region marked by #pragma auto_inline(off) and #pragma auto_inline(on) directives. For information on another way to provide inlining hints to the compiler, see the #pragma intrinsic directive.

> ⓘ **Note**

Information that is gathered from profiling test runs overrides optimizations that would otherwise be in effect because you specified **/Ob**, **/Os**, or **/Ot**. For more information, see **Profile-Guided Optimizations**.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Optimization** property page.
3. Modify the **Inline Function Expansion** property.

The **/Ob3** option isn't available in the **Inline Function Expansion** property. To set **/Ob3**:

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter **/Ob3** in **Additional Options**.

To set this compiler option programmatically

- See InlineFunctionExpansion.

See also

/O Options (Optimize Code)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Od (Disable (Debug))

Article • 08/03/2021

Turns off all optimizations in the program and speeds compilation.

Syntax

```
/Od
```

Remarks

This option is the default. Because **/Od** suppresses code movement, it simplifies the debugging process. For more information about compiler options for debugging, see /Z7, /Zi, /ZI (Debug Information Format).

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Optimization** property page.
3. Modify the **Optimization** property.

To set this compiler option programmatically

- See Optimization.

See also

/O Options (Optimize Code)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
/Z7, /Zi, /ZI (Debug Information Format)

`/Og` (Global Optimizations)

Article • 10/20/2021

Deprecated. Provides local and global optimizations, automatic-register allocation, and loop optimization. We recommend you use either /O1 (Minimize Size) or /O2 (Maximize Speed) instead.

Syntax

> `/Og`

Remarks

`/Og` is deprecated. These optimizations are now enabled by default when any optimizations are enabled. For more information on optimizations, see /O1, /O2 (Minimize Size, Maximize Speed), or /Ox (Enable Most Speed Optimizations).

The following optimizations are available under `/Og`:

- Local and global common subexpression elimination

 In this optimization, the value of a common subexpression is calculated once. In the following example, if the values of `b` and `c` don't change between the three expressions, the compiler can assign the calculation of `b + c` to a temporary variable, and use that variable for `b + c`:

  ```
  C

  a = b + c;
  d = b + c;
  e = b + c;
  ```

 For local common subexpression optimization, the compiler examines short sections of code for common subexpressions. For global common subexpression optimization, the compiler searches entire functions for common subexpressions.

- Automatic register allocation

 This optimization allows the compiler to store frequently used variables and subexpressions in registers. The `register` keyword is ignored by default, and causes a diagnostic under `/std:c++17` or later.

- Loop optimization

 This optimization removes invariant subexpressions from the body of a loop. An optimal loop contains only expressions whose values change through each execution of the loop. In the following example, the expression `x + y` doesn't change in the loop body:

  ```
  C

  i = -100;
  while( i < 0 ) {
      i += x + y;
  }
  ```

 After optimization, `x + y` is calculated once rather than every time the loop is executed:

  ```
  C

  i = -100;
  t = x + y;
  while( i < 0 ) {
      i += t;
  }
  ```

 Loop optimization is much more effective when the compiler can assume no aliasing, which you set with __restrict, noalias, or restrict.

 > **Note**
 >
 > You can enable or disable global optimization on a function-by-function basis using the `optimize` pragma together with the `g` option.

For related information, see /Oi (Generate intrinsic functions) and /Ox (Enable most speed optimizations).

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Command-Line Syntax

/Oi (Generate Intrinsic Functions)

Article • 08/03/2021

Replaces some function calls with intrinsic or otherwise special forms of the function that help your application run faster.

Syntax

```
/Oi[-]
```

Remarks

Programs that use intrinsic functions are faster because they do not have the overhead of function calls, but may be larger because of the additional code created.

See intrinsic for more information on which functions have intrinsic forms.

/Oi is only a request to the compiler to replace some function calls with intrinsics; the compiler may call the function (and not replace the function call with an intrinsic) if it will result in better performance.

x86 Specific

The intrinsic floating-point functions do not perform any special checks on input values and so work in restricted ranges of input, and have different exception handling and boundary conditions than the library routines with the same name. Using the true intrinsic forms implies loss of IEEE exception handling, and loss of `_matherr` and `errno` functionality; the latter implies loss of ANSI conformance. However, the intrinsic forms can considerably speed up floating-point-intensive programs, and for many programs, the conformance issues are of little practical value.

You can use the Za compiler option to override generation of true intrinsic floating-point options. In this case, the functions are generated as library routines that pass arguments directly to the floating-point chip instead of pushing them onto the program stack.

END x86 Specific

You also use intrinsic to create intrinsic functions, or function (C/C++) to explicitly force a function call.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Optimization** property page.
3. Modify the **Enable Intrinsic Functions** property.

To set this compiler option programmatically

- See EnableIntrinsicFunctions.

See also

/O Options (Optimize Code)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Compiler Intrinsics

`/Os`, `/Ot` (Favor Small Code, Favor Fast Code)

Article • 04/19/2022

The `/Os` and `/Ot` compiler options specify whether to favor size (`/Os`) or speed (`/Ot`) when optimizing code.

Syntax

> `/Os`
> `/Ot`

Remarks

`/Os` (Favor Small Code) minimizes the size of EXEs and DLLs by instructing the compiler to favor size over speed. The compiler can reduce many C and C++ constructs to functionally similar sequences of machine code. Occasionally these differences offer tradeoffs of size versus speed. The `/Os` and `/Ot` options allow you to specify a preference for one over the other:

`/Ot` (Favor Fast Code) maximizes the speed of EXEs and DLLs by instructing the compiler to favor speed over size. `/Ot` is the default when optimizations are enabled. The compiler can reduce many C and C++ constructs to functionally similar sequences of machine code. Occasionally, these differences offer tradeoffs of size versus speed. The `/Ot` option is implied by the /O2 (Maximize speed) option. The `/O2` option combines several options to produce faster code.

> ⓘ **Note**
>
> Information that's gathered from profiling test runs overrides any optimizations that would otherwise be in effect if you specify `/Ob`, `/Os`, or `/Ot`. For more information, see **Profile-Guided Optimizations**.

x86-specific example

The following example code demonstrates the difference between the `/Os` (Favor small code) option and the `/Ot` (Favor fast code) option:

> **Note**
>
> This example describes the expected behavior when using `/Os` or `/Ot`. However, compiler behavior from release to release may result in different optimizations for the code below.

C

```
/* differ.c
  This program implements a multiplication operator
  Compile with /Os to implement multiply explicitly as multiply.
  Compile with /Ot to implement as a series of shift and LEA instructions.
*/
int differ(int x)
{
    return x * 71;
}
```

As shown in the fragment of machine code below, when *`differ.c`* is compiled for size (`/Os`), the compiler implements the multiply expression in the return statement explicitly as a multiply to produce a short but slower sequence of code:

asm

```
mov    eax, DWORD PTR _x$[ebp]
imul   eax, 71                  ; 00000047H
```

Alternately, when *`differ.c`* is compiled for speed (`/Ot`), the compiler implements the multiply expression in the return statement as a series of shift and `LEA` instructions to produce a fast but longer sequence of code:

asm

```
mov    eax, DWORD PTR _x$[ebp]
mov    ecx, eax
shl    eax, 3
lea    eax, DWORD PTR [eax+eax*8]
sub    eax, ecx
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Optimization** property page.
3. Modify the **Favor Size or Speed** property.

To set this compiler option programmatically

- See FavorSizeOrSpeed.

See also

/O options (Optimize code)
MSVC compiler options
MSVC compiler command-line syntax

`/Ox` (Enable Most Speed Optimizations)

Article • 08/03/2021

The `/Ox` compiler option enables a combination of optimizations that favor speed. In some versions of the Visual Studio IDE and the compiler help message, it's called *full optimization*, but the `/Ox` compiler option enables only a subset of the speed optimization options enabled by `/O2`.

Syntax

> `/Ox`

Remarks

The `/Ox` compiler option enables the `/O` compiler options that favor speed. The `/Ox` compiler option doesn't include the additional /GF (Eliminate Duplicate Strings) and /Gy (Enable Function-Level Linking) options enabled by /O1 or /O2 (Minimize Size, Maximize Speed). The additional options applied by `/O1` and `/O2` can cause pointers to strings or to functions to share a target address, which can affect debugging and strict language conformance. The `/Ox` option is an easy way to enable most optimizations without including `/GF` and `/Gy`. For more information, see the descriptions of the /GF and /Gy options.

The `/Ox` compiler option is the same as using the following options in combination:

- /Ob (Inline Function Expansion), where the option parameter is 2 (`/Ob2`)
- /Oi (Generate Intrinsic Functions)
- /Ot (Favor Fast Code)
- /Oy (Frame-Pointer Omission)

`/Ox` is mutually exclusive from:

- /O1 (Minimize Size)
- /O2 (Maximize Speed)
- /Od (Disable (Debug))

You can cancel the bias toward speed of the `/0x` compiler option if you specify `/0xs`, which combines the `/0x` compiler option with /Os (Favor Small Code). The combined options favor smaller code size. The `/0xs` option is exactly the same as specifying `/0x` `/0s` when the options appear in that order.

To apply all available file-level optimizations for release builds, we recommend you specify /O2 (Maximize Speed) instead of `/0x`, and /O1 (Minimize Size) instead of `/0xs`. For even more optimization in release builds, also consider the /GL (Whole Program Optimization) compiler option and /LTCG (Link-time Code Generation) linker option.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Optimization** property page.
3. Modify the **Optimization** property.

To set this compiler option programmatically

- See Optimization.

See also

/O Options (Optimize Code)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Oy (Frame-Pointer Omission)

Article • 08/03/2021

Suppresses creation of frame pointers on the call stack.

Syntax

> /Oy[-]

Remarks

This option speeds function calls, because no frame pointers need to be set up and removed. It also frees one more register for general usage.

/Oy enables frame-pointer omission and **/Oy-** disables omission. In x64 compilers, **/Oy** and **/Oy-** are not available.

If your code requires frame-based addressing, you can specify the **/Oy-** option after the **/Ox** option or use optimize with the "**y**" and **off** arguments to gain maximum optimization with frame-based addressing. The compiler detects most situations where frame-based addressing is required (for instance, with the `_alloca` and `setjmp` functions and with structured exception handling).

The /Ox (Enable Most Speed Optimizations) and /O1, /O2 (Minimize Size, Maximize Speed) options imply **/Oy**. Specifying **/Oy-** after the **/Ox**, **/O1**, or **/O2** option disables **/Oy**, whether it is explicit or implied.

The **/Oy** compiler option makes using the debugger more difficult because the compiler suppresses frame pointer information. If you specify a debug compiler option (/Z7, /Zi, /ZI), we recommend that you specify the **/Oy-** option after any other optimization compiler options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Optimization** property page.

3. Modify the **Omit Frame Pointers** property. This property adds or removes only the **/Oy** option. If you want to add the **/Oy-** option, select the **Command Line** property page and modify **Additional options**.

To set this compiler option programmatically

- See OmitFramePointers.

See also

/O Options (Optimize Code)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/openmp` (Enable OpenMP Support)

Article • 07/05/2023

Causes the compiler to process #pragma omp directives in support of OpenMP.

Syntax

> `/openmp`
>
> `/openmp:experimental`
>
> `/openmp:llvm`

Remarks

`#pragma omp` is used to specify Directives and Clauses. If `/openmp` isn't specified in a compilation, the compiler ignores OpenMP clauses and directives. OpenMP Function calls are processed by the compiler even if `/openmp` isn't specified.

The C++ compiler currently supports the OpenMP 2.0 standard. Visual Studio 2019 also now offers SIMD functionality. To use SIMD, compile using the `/openmp:experimental` option. This option enables both the usual OpenMP features, and OpenMP SIMD features not available when using the `/openmp` switch.

Starting in Visual Studio 2019 version 16.9, you can use the experimental `/openmp:llvm` option instead of `/openmp` to target the LLVM OpenMP runtime. Support currently isn't available for production code, since the required libomp DLLs aren't redistributable. The option supports the same OpenMP 2.0 directives as `/openmp`. And, it supports all the SIMD directives supported by the `/openmp:experimental` option. It also supports unsigned integer indices in parallel for loops according to the OpenMP 3.0 standard. For more information, see Improved OpenMP Support for C++ in Visual Studio.

The `/openmp:llvm` option supports the x64 architecture. Starting with Visual Studio 2019 version 16.10, it also supports the x86 and ARM64 architectures. This option isn't compatible with `/clr` or `/ZW`.

Applications compiled by using both `/openmp` and `/clr` can only be run in a single application domain process. Multiple application domains aren't supported. That is, when the module constructor (`.cctor`) is run, it detects if the process is compiled using `/openmp`, and if the app is loaded into a non-default runtime. For more information, see

appdomain, /clr (Common Language Runtime Compilation), and Initialization of Mixed Assemblies.

If you attempt to load an app compiled using both `/openmp` and `/clr` into a non-default application domain, a TypeInitializationException exception is thrown outside the debugger, and a `OpenMPWithMultipleAppdomainsException` exception is thrown in the debugger.

These exceptions can also be raised in the following situations:

- If your application is compiled using `/clr` but not `/openmp`, and is loaded into a non-default application domain, where the process includes an app compiled using `/openmp`.
- If you pass your `/clr` app to a utility, such as regasm.exe, which loads its target assemblies into a non-default application domain.

The common language runtime's code access security doesn't work in OpenMP regions. If you apply a CLR code access security attribute outside a parallel region, it won't be in effect in the parallel region.

Microsoft doesn't recommend that you write `/openmp` apps that allow partially trusted callers. Don't use AllowPartiallyTrustedCallersAttribute, or any CLR code access security attributes.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Expand the **Configuration Properties** > **C/C++** > **Language** property page.
3. Modify the **OpenMP Support** property.

To set this compiler option programmatically

- See OpenMP.

Example

The following sample shows some of the effects of thread pool startup versus using the thread pool after it has started. Assuming an x64, single core, dual processor, the thread pool takes about 16 ms to start up. After that, there's little extra cost for the thread pool.

When you compile using `/openmp`, the second call to test2 never runs any longer than if you compile using `/openmp-`, as there's no thread pool startup. At a million iterations, the `/openmp` version is faster than the `/openmp-` version for the second call to test2. At 25 iterations, both `/openmp-` and `/openmp` versions register less than the clock granularity.

If you have only one loop in your application and it runs in less than 15 ms (adjusted for the approximate overhead on your machine), `/openmp` may not be appropriate. If it's higher, you may want to consider using `/openmp`.

C++

```
// cpp_compiler_options_openmp.cpp
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

volatile DWORD dwStart;
volatile int global = 0;

double test2(int num_steps) {
   int i;
   global++;
   double x, pi, sum = 0.0, step;

   step = 1.0 / (double) num_steps;

   #pragma omp parallel for reduction(+:sum) private(x)
   for (i = 1; i <= num_steps; i++) {
      x = (i - 0.5) * step;
      sum = sum + 4.0 / (1.0 + x*x);
   }

   pi = step * sum;
   return pi;
}

int main(int argc, char* argv[]) {
   double   d;
   int n = 1000000;

   if (argc > 1)
      n = atoi(argv[1]);

   dwStart = GetTickCount();
```

```
   d = test2(n);
   printf_s("For %d steps, pi = %.15f, %d milliseconds\n", n, d,
GetTickCount() - dwStart);

   dwStart = GetTickCount();
   d = test2(n);
   printf_s("For %d steps, pi = %.15f, %d milliseconds\n", n, d,
GetTickCount() - dwStart);
}
```

See also

MSVC compiler options

MSVC compiler command-line syntax

OpenMP in MSVC

`/options:strict` (Unrecognized compiler options are errors)

Article • 01/10/2022

The `/options:strict` compiler option tells the compiler to return an error code if a compiler option isn't recognized.

Syntax

```
/options:strict
```

Remarks

The `/options:strict` compiler option causes the compiler driver cl.exe to exit with an error code after all command-line options are parsed if another command-line option or argument isn't recognized. The compiler emits error D8043 for any command-line option or argument that isn't recognized.

The `/options:strict` option is available starting in Visual Studio 2022 version 17.0. In earlier versions of the compiler, or if `/options:strict` isn't specified, the compiler doesn't exit on an unrecognized option. It emits warning D9002, ignores the unrecognized option, and continues processing.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Add `/options:strict` to the **Additional options:** pane.

See also

/Zc (Conformance)

/P (Preprocess to a File)

Article • 08/03/2021

Preprocesses C and C++ source files and writes the preprocessed output to a file.

Syntax

```
/P
```

Remarks

The file has the same base name as the source file and an .i extension. In the process, all preprocessor directives are carried out, macro expansions are performed, and comments are removed. To preserve comments in the preprocessed output, use the /C (Preserve Comments During Preprocessing) option along with **/P**.

/P adds `#line` directives to the output, at the beginning and end of each included file and around lines removed by preprocessor directives for conditional compilation. These directives renumber the lines of the preprocessed file. As a result, errors generated during later stages of processing refer to the line numbers of the original source file rather than lines in the preprocessed file. To suppress the generation of `#line` directives, use /EP (Preprocess to stdout Without #line Directives) as well as **/P**.

The **/P** option suppresses compilation. It does not produce an .obj file, even if you use /Fo (Object File Name). You must resubmit the preprocessed file for compilation. **/P** also suppresses the output files from the **/FA**, **/Fa**, and **/Fm** options. For more information, see /FA, /Fa (Listing File) and /Fm (Name Mapfile).

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Preprocessor** property page.
3. Modify the **Generate Preprocessed File** property.

To set this compiler option programmatically

- See GeneratePreprocessedFile.

Example

The following command line preprocesses `ADD.C`, preserves comments, adds `#line` directives, and writes the result to a file, `ADD.I`:

```
CL /P /C ADD.C
```

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
/Fi (Preprocess Output File Name)

`/permissive-` (Standards conformance)

Article • 10/12/2023

Specify standards conformance mode to the compiler. Use this option to help you identify and fix conformance issues in your code, to make it both more correct and more portable.

Syntax

> `/permissive-`
>
> `/permissive`

Remarks

The `/permissive-` option is supported in Visual Studio 2017 and later. `/permissive` is supported in Visual Studio 2019 version 16.8 and later.

You can use the `/permissive-` compiler option to specify standards-conforming compiler behavior. This option disables permissive behaviors, and sets the /Zc compiler options for strict conformance. In the IDE, this option also makes the IntelliSense engine underline non-conforming code.

The `/permissive-` option uses the conformance support in the current compiler version to determine which language constructs are non-conforming. The option doesn't determine if your code conforms to a specific version of the C++ standard. To enable all implemented compiler support for the latest draft standard, use the /std:c++latest option. To restrict the compiler support to the currently implemented C++20 standard, use the /std:c++20 option. To restrict the compiler support to the currently implemented C++17 standard, use the /std:c++17 option. To restrict the compiler support to more closely match the C++14 standard, use the /std:c++14 option, which is the default.

The `/permissive-` option is implicitly set by the `/std:c++latest` option starting in Visual Studio 2019 version 16.8, and in version 16.11 by the `/std:c++20` option. `/permissive-` is required for C++20 Modules support. Perhaps your code doesn't need modules support but requires other features enabled under `/std:c++20` or `/std:c++latest`. You can explicitly enable Microsoft extension support by using the `/permissive` option without the trailing dash. The `/permissive` option must come after any option that sets `/permissive-` implicitly.

By default, the `/permissive-` option is set in new projects created by Visual Studio 2017 version 15.5 and later versions. It's not set by default in earlier versions. When the option is set, the compiler generates diagnostic errors or warnings when non-standard language constructs are detected in your code. These constructs include some common bugs in pre-C++11 code.

The `/permissive-` option is compatible with almost all of the header files from the latest Windows Kits, such as the Software Development Kit (SDK) or Windows Driver Kit (WDK), starting in the Windows Fall Creators SDK (10.0.16299.0). Older versions of the SDK may fail to compile under `/permissive-` for various source code conformance reasons. The compiler and SDKs ship on different release timelines, so there are some remaining issues. For specific header file issues, see Windows header issues below.

The `/permissive-` option sets the /Zc:referenceBinding, /Zc:strictStrings, and /Zc:rvalueCast options to conforming behavior. These options default to non-conforming behavior. You can pass specific `/Zc` options after `/permissive-` on the command line to override this behavior.

In versions of the compiler beginning in Visual Studio 2017 version 15.3, the `/permissive-` option sets the /Zc:ternary option. The compiler also implements more of the requirements for two-phase name look-up. When the `/permissive-` option is set, the compiler parses function and class template definitions, and identifies dependent and non-dependent names used in the templates. In this release, only name dependency analysis is performed.

As of Visual Studio 2022 Update 17.6, the `/permissive-` option sets the /Zc:lambda and /Zc:externConstexpr options. In prior versions, `/permissive-` didn't set either one.

Environment-specific extensions and language areas that the standard leaves up to the implementation aren't affected by `/permissive-`. For example, the Microsoft-specific `__declspec`, calling convention and structured exception handling keywords, and compiler-specific `pragma` directives or attributes aren't flagged by the compiler in `/permissive-` mode.

The MSVC compiler in earlier versions of Visual Studio 2017 doesn't support all C++11, C++14, or C++17 standards-conforming code. Depending on the version of Visual Studio, the `/permissive-` option may not detect issues in some aspects of two-phase name lookup, binding a non-const reference to a temporary, treating copy init as direct init, allowing multiple user-defined conversions in initialization, or alternative tokens for logical operators, and other non-supported conformance areas. For more information about conformance issues in Visual C++, see Nonstandard Behavior. To get the most out of `/permissive-`, update Visual Studio to the latest version.

How to fix your code

Here are some examples of code that is detected as non-conforming when you use `/permissive-`, along with suggested ways to fix the issues.

Use `default` as an identifier in native code

C++

```
void func(int default); // Error C2321: 'default' is a keyword, and
                        // cannot be used in this context
```

Look up members in dependent base

C++

```
template <typename T>
struct B
{
    void f() {}
    template <typename U>
    struct S { void operator()(){ return; } };
};

template <typename T>
struct D : public B<T> // B is a dependent base because its type
                       // depends on the type of T.
{
    // One possible fix for non-template members and function
    // template members is a using statement:
    // using B<T>::f;
    // If it's a type, don't forget the 'typename' keyword.

    void g()
    {
        f(); // error C3861: 'f': identifier not found
        // Another fix is to change the call to 'this->f();'
    }

    void h()
    {
        S<int> s; // C2065 or C3878
        // Since template S is dependent, the type must be qualified
        // with the `typename` keyword.
        // To fix, replace the declaration of s with:
        // typename B<T>::template S<int> s;
        // Or, use this:
        // typename D::template S<int> s;
        s();
```

```
    }
};

void h() {
    D<int> d;
    d.g();
    d.h();
}
```

Use of qualified names in member declarations

C++

```
struct A {
    void A::f() { } // error C4596: illegal qualified name in member
                    // declaration.
                    // Remove redundant 'A::' to fix.
};
```

Initialize multiple union members in a member initializer

C++

```
union U
{
    U()
        : i(1), j(1) // error C3442: Initializing multiple members of
                     // union: 'U::i' and 'U::j'.
                     // Remove all but one of the initializations to fix.
    {}
    int i;
    int j;
};
```

Hidden friend name lookup rules

A declaration outside a class can make a hidden friend visible:

C++

```
// Example 1
struct S {
    friend void f(S *);
};
// Uncomment this declaration to make the hidden friend visible:
// void f(S *); // This declaration makes the hidden friend visible
```

```
using type = void (*)(S *);
type p = &f; // error C2065: 'f': undeclared identifier.
```

Use of literal `nullptr` can prevent argument dependent lookup:

C++

```
// Example 2
struct S {
    friend void f(S *);
};
void g() {
    // Using nullptr instead of S prevents argument dependent lookup in S
    f(nullptr); // error C3861: 'f': identifier not found

    S *p = nullptr;
    f(p); // Hidden friend now found via argument-dependent lookup.
}
```

You can enable the hidden friend name lookup rules independently of **`/permissive`** by using /Zc:hiddenFriend. If you want legacy behavior for hidden friend name lookup, but otherwise want **`/permissive-`** behavior, use the **`/Zc:hiddenFriend-`** option.

Use scoped enums in array bounds

C++

```
enum class Color {
    Red, Green, Blue
};

int data[Color::Blue]; // error C3411: 'Color' is not valid as the size
                       // of an array as it is not an integer type.
                       // Cast to type size_t or int to fix.
```

Use for each in native code

C++

```
void func() {
    int array[] = {1, 2, 30, 40};
    for each (int i in array) // error C4496: nonstandard extension
                              // 'for each' used: replace with
                              // ranged-for statement:
                              // for (int i: array)
    {
        // ...
```

```
    }
}
```

Use of ATL attributes

Microsoft-specific ATL attributes can cause issues under `/permissive-`:

C++

```
// Example 1
[uuid("594382D9-44B0-461A-8DE3-E06A3E73C5EB")]
class A {};
```

You can fix the issue by using the `__declspec` form instead:

C++

```
// Fix for example 1
class __declspec(uuid("594382D9-44B0-461A-8DE3-E06A3E73C5EB")) B {};
```

A more complex example:

C++

```
// Example 2
[emitidl];
[module(name="Foo")];

[object, local, uuid("9e66a290-4365-11d2-a997-00c04fa37ddb")]
__interface ICustom {
    HRESULT Custom([in] longl, [out, retval] long*pLong);
    [local] HRESULT CustomLocal([in] longl, [out, retval] long*pLong);
};

[coclass, appobject, uuid("9e66a294-4365-11d2-a997-00c04fa37ddb")]
class CFoo : public ICustom
{};
```

Resolution requires extra build steps. In this case, create an IDL file:

C++

```
// Fix for example 2
// First, create the *.idl file. The vc140.idl generated file can be
// used to automatically obtain a *.idl file for the interfaces with
// annotation. Second, add a midl step to your build system to make
// sure that the C++ interface definitions are outputted.
```

```
// Last, adjust your existing code to use ATL directly as shown in
// the atl implementation section.

-- IDL  FILE--
import "docobj.idl";

[object, local, uuid(9e66a290-4365-11d2-a997-00c04fa37ddb)]
interface ICustom : IUnknown {
    HRESULT Custom([in] longl, [out,retval] long*pLong);
    [local] HRESULT CustomLocal([in] longl, [out,retval] long*pLong);
};

[ version(1.0), uuid(29079a2c-5f3f-3325-99a1-3ec9c40988bb) ]
library Foo {
    importlib("stdole2.tlb");
    importlib("olepro32.dll");

    [version(1.0), appobject, uuid(9e66a294-4365-11d2-a997-00c04fa37ddb)]
    coclass CFoo { interface ICustom; };
}

-- ATL IMPLEMENTATION--
#include <idl.header.h>
#include <atlbase.h>

class ATL_NO_VTABLE CFooImpl : public ICustom,
    public ATL::CComObjectRootEx<CComMultiThreadModel>
{
    public:BEGIN_COM_MAP(CFooImpl)
    COM_INTERFACE_ENTRY(ICustom)
    END_COM_MAP()
};
```

Ambiguous conditional operator arguments

In versions of the compiler before Visual Studio 2017 version 15.3, the compiler accepted arguments to the conditional operator (or ternary operator) `?:` that are considered ambiguous by the Standard. In `/permissive-` mode, the compiler now issues one or more diagnostics in cases that compiled without diagnostics in earlier versions.

Common errors that may result from this change include:

- `error C2593: 'operator ?' is ambiguous`
- `error C2679: binary '?': no operator found which takes a right-hand operand of type 'B' (or there is no acceptable conversion)`
- `error C2678: binary '?': no operator found which takes a left-hand operand of type 'A' (or there is no acceptable conversion)`

- `error C2446: ':': no conversion from 'B' to 'A'`

A typical code pattern that can cause this issue is when some class `C` provides both a non-explicit constructor from another type `T` and a non-explicit conversion operator to type `T`. The conversion of the second argument to the third argument's type is a valid conversion. So is the conversion of the third argument to the second argument's type. Since both are valid, it's ambiguous according to the standard.

C++

```
// Example 1: class that provides conversion to and initialization from some
type T
struct A
{
    A(int);
    operator int() const;
};

extern bool cond;

A a(42);
// Accepted when /Zc:ternary or /permissive- is not used:
auto x = cond ? 7 : a; // A: permissive behavior prefers A(7) over (int)a
// Accepted always:
auto y = cond ? 7 : int(a);
auto z = cond ? A(7) : a;
```

There's an important exception to this common pattern when T represents one of the null-terminated string types (for example, `const char *`, `const char16_t *`, and so on) and the actual argument to `?:` is a string literal of corresponding type. C++17 has changed semantics from C++14. As a result, the code in example 2 is accepted under **`/std:c++14`** and rejected under **`/std:c++17`** or later when **`/Zc:ternary`** or **`/permissive-`** is used.

C++

```
// Example 2: exception from the above
struct MyString
{
    MyString(const char* s = "") noexcept;  // from char*
    operator const char* () const noexcept; //   to char*
};

extern bool cond;

MyString s;
// Using /std:c++14, /permissive- or /Zc:ternary behavior
// is to prefer MyString("A") over (const char*)s
```

```
// but under /std:c++17 this line causes error C2445:
auto x = cond ? "A" : s;
// You can use a static_cast to resolve the ambiguity:
auto y = cond ? "A" : static_cast<const char*>(s);
```

You may also see errors in conditional operators with one argument of type `void`. This case may be common in ASSERT-like macros.

C++

```
// Example 3: void arguments
void myassert(const char* text, const char* file, int line);
// Accepted when /Zc:ternary or /permissive- is not used:
#define ASSERT_A(ex) (void)((ex) ? 1 : myassert(#ex, __FILE__, __LINE__))
// Accepted always:
#define ASSERT_B(ex) (void)((ex) ? void() : myassert(#ex, __FILE__,
__LINE__))
```

You may also see errors in template metaprogramming, where conditional operator result types may change under `/Zc:ternary` and `/permissive-`. One way to resolve this issue is to use std::remove_reference on the resulting type.

C++

```
// Example 4: different result types
extern bool cond;
extern int count;
char  a = 'A';
const char  b = 'B';
decltype(auto) x = cond ? a : b; // char without, const char& with
/Zc:ternary
const char (&z)[2] = count > 3 ? "A" : "B"; // const char* without
/Zc:ternary
```

Two-phase name look-up

When the `/permissive-` option is set, the compiler parses function and class template definitions, identifying dependent and non-dependent names used in templates as required for two-phase name look-up. In Visual Studio 2017 version 15.3, name dependency analysis is performed. In particular, non-dependent names that aren't declared in the context of a template definition cause a diagnostic message as required by the ISO C++ standards. In Visual Studio 2017 version 15.7, binding of non-dependent names that require argument-dependent look-up in the definition context is also done.

C++

```
// dependent base
struct B {
    void g() {}
};

template<typename T>
struct D : T {
    void f() {
        // The call to g was incorrectly allowed in VS2017:
        g();  // Now under /permissive-: C3861
        // Possible fixes:
        // this->g();
        // T::g();
    }
};

int main()
{
    D<B> d;
    d.f();
}
```

If you want legacy behavior for two-phase lookup, but otherwise want `/permissive-` behavior, add the `/Zc:twoPhase-` option.

Windows header issues

The `/permissive-` option is too strict for versions of the Windows Kits before Windows Fall Creators Update SDK (10.0.16299.0), or the Windows Driver Kit (WDK) version 1709. We recommend you update to the latest versions of the Windows Kits to use `/permissive-` in your Windows or device driver code.

Certain header files in the Windows April 2018 Update SDK (10.0.17134.0), the Windows Fall Creators Update SDK (10.0.16299.0), or the Windows Driver Kit (WDK) 1709, still have issues that make them incompatible with use of `/permissive-`. To work around these issues, we recommend you restrict the use of these headers to only those source code files that require them, and remove the `/permissive-` option when you compile those specific source code files.

These WinRT WRL headers released in the Windows April 2018 Update SDK (10.0.17134.0) aren't clean with `/permissive-`. To work around these issues, either don't use `/permissive-`, or use `/permissive-` with `/Zc:twoPhase-` when you work with these headers:

- Issues in *`winrt/wrl/async.h`*

Output

```
C:\Program Files (x86)\Windows
Kits\10\Include\10.0.17134.0\winrt\wrl\async.h(483): error C3861:
'TraceDelegateAssigned': identifier not found
C:\Program Files (x86)\Windows
Kits\10\Include\10.0.17134.0\winrt\wrl\async.h(491): error C3861:
'CheckValidStateForDelegateCall': identifier not found
C:\Program Files (x86)\Windows
Kits\10\Include\10.0.17134.0\winrt\wrl\async.h(509): error C3861:
'TraceProgressNotificationStart': identifier not found
C:\Program Files (x86)\Windows
Kits\10\Include\10.0.17134.0\winrt\wrl\async.h(513): error C3861:
'TraceProgressNotificationComplete': identifier not found
```

- Issue in `winrt/wrl/implements.h`

Output

```
C:\Program Files (x86)\Windows
Kits\10\include\10.0.17134.0\winrt\wrl\implements.h(2086): error C2039:
'SetStrongReference': is not a member of
'Microsoft::WRL::Details::WeakReferenceImpl'
```

These User Mode headers released in the Windows April 2018 Update SDK (10.0.17134.0) aren't clean with `/permissive-`. To work around these issues, don't use `/permissive-` when working with these headers:

- Issues in `um/Tune.h`

Output

```
C:\ProgramFiles(x86)\Windows
Kits\10\include\10.0.17134.0\um\tune.h(139): error C3861: 'Release':
identifier not found
C:\Program Files (x86)\Windows
Kits\10\include\10.0.17134.0\um\tune.h(559): error C3861: 'Release':
identifier not found
C:\Program Files (x86)\Windows
Kits\10\include\10.0.17134.0\um\tune.h(1240): error C3861: 'Release':
identifier not found
C:\Program Files (x86)\Windows
Kits\10\include\10.0.17134.0\um\tune.h(1240): note: 'Release': function
declaration must be available as none of the arguments depend on a
template parameter
```

- Issue in `um/spddkhlp.h`

Output

```
C:\Program Files (x86)\Windows
Kits\10\include\10.0.17134.0\um\spddkhlp.h(759): error C3861: 'pNode':
identifier not found
```

- Issues in *um/refptrco.h*

Output

```
C:\Program Files (x86)\Windows
Kits\10\include\10.0.17134.0\um\refptrco.h(179): error C2760: syntax
error: unexpected token 'identifier', expected 'type specifier'
C:\Program Files (x86)\Windows
Kits\10\include\10.0.17134.0\um\refptrco.h(342): error C2760: syntax
error: unexpected token 'identifier', expected 'type specifier'
C:\Program Files (x86)\Windows
Kits\10\include\10.0.17134.0\um\refptrco.h(395): error C2760: syntax
error: unexpected token 'identifier', expected 'type specifier'
```

These issues are specific to User Mode headers in the Windows Fall Creators Update SDK (10.0.16299.0):

- Issue in *um/Query.h*

 When you use the **/permissive-** compiler switch, the `tagRESTRICTION` structure doesn't compile because of the `case(RTOr)` member `or`.

C++

```
struct tagRESTRICTION
{
     ULONG rt;
     ULONG weight;
     /* [switch_is][switch_type] */ union _URes
     {
         /* [case()] */ NODERESTRICTION ar;
         /* [case()] */ NODERESTRICTION or;  // error C2059: syntax
error: '||'
         /* [case()] */ NODERESTRICTION pxr;
         /* [case()] */ VECTORRESTRICTION vr;
         /* [case()] */ NOTRESTRICTION nr;
         /* [case()] */ CONTENTRESTRICTION cr;
         /* [case()] */ NATLANGUAGERESTRICTION nlr;
         /* [case()] */ PROPERTYRESTRICTION pr;
         /* [default] */  /* Empty union arm */
     } res;
};
```

To address this issue, compile files that include *Query.h* without the **/permissive-** option.

- Issue in *um/cellularapi_oem.h*

 When you use the **/permissive-** compiler switch, the forward declaration of `enum UICCDATASTOREACCESSMODE` causes a warning:

  ```
  C++

  typedef enum UICCDATASTOREACCESSMODE UICCDATASTOREACCESSMODE; // C4471
  ```

 The forward declaration of an unscoped `enum` is a Microsoft extension. To address this issue, compile files that include *cellularapi_oem.h* without the **/permissive-** option, or use the /wd option to silence warning C4471.

- Issue in *um/omscript.h*

 In C++03, a conversion from a string literal to `BSTR` (which is a typedef to `wchar_t *`) is deprecated but allowed. In C++11, the conversion is no longer allowed.

  ```
  C++

  virtual /* [id] */ HRESULT STDMETHODCALLTYPE setExpression(
       /* [in] */ __RPC__in BSTR propname,
       /* [in] */ __RPC__in BSTR expression,
       /* [in][defaultvalue] */ __RPC__in BSTR language = L"") = 0; //
  C2440
  ```

 To address this issue, compile files that include omscript.h without the **/permissive-** option, or use **/Zc:strictStrings-** instead.

To set this compiler option in the Visual Studio development environment

In Visual Studio 2017 version 15.5 and later versions, use this procedure:

1. Open your project's **Property Pages** dialog box.
2. Select the **Configuration Properties** > **C/C++** > **Language** property page.
3. Change the **Conformance mode** property value to **Yes (/permissive-)**. Choose **OK** or **Apply** to save your changes.

In versions before Visual Studio 2017 version 15.5, use this procedure:

1. Open your project's **Property Pages** dialog box.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the **/permissive-** compiler option in the **Additional Options** box. Choose **OK** or **Apply** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Q Options (Low-Level Operations)

Article • 08/03/2021

You can use the **/Q** compiler options to perform the following low-level compiler operations:

- /Qfast_transcendentals (Force Fast Transcendentals): Generates fast transcendentals.
- /QIfist (Suppress _ftol): Suppresses `_ftol` when a conversion from a floating-point type to an integer type is required (x86 only).
- /Qimprecise_fwaits (Remove fwaits Inside Try Blocks): Removes `fwait` commands inside `try` blocks.
- /QIntel-jcc-erratum: Mitigates the performance impact caused by the Intel Jump Conditional Code (JCC) erratum microcode update.
- /Qpar (Auto-Parallelizer): Enables automatic parallelization of loops that are marked with the #pragma loop() directive.
- /Qpar-report (Auto-Parallelizer Reporting Level): Enables reporting levels for automatic parallelization.
- /Qsafe_fp_loads: Suppresses optimizations for floating-point register loads and for moves between memory and MMX registers.
- /Qspectre: Generates instructions to mitigate certain Spectre security vulnerabilities.
- /Qspectre-load: Generates instructions to mitigate Spectre security vulnerabilities based on loads.
- /Qspectre-load-cf: Generates instructions to mitigate Spectre security vulnerabilities based on control flow instructions which load.
- /Qvec-report (Auto-Vectorizer Reporting Level): Enables reporting levels for automatic vectorization.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/Qfast_transcendentals` (Force fast transcendentals)

Article • 03/03/2022

Generates inline code for transcendental functions.

Syntax

```
/Qfast_transcendentals
```

Remarks

This compiler option forces transcendental functions to be converted to inline code to improve execution speed. This option has an effect only when paired with `/fp:except` or `/fp:precise`. Generating inline code for transcendental functions is already the default behavior under `/fp:fast`.

This option is incompatible with `/fp:strict`. For more information about floating point compiler options, see /fp (Specify Floating-Point Behavior).

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/Q Options (Low-level operations)
MSVC compiler options

MSVC compiler command-line syntax

/QIfist (Suppress _ftol)

Article • 08/03/2021

Deprecated. Suppresses the call of the helper function `_ftol` when a conversion from a floating-point type to an integral type is required.

Syntax

```
/QIfist
```

Remarks

> **Note**
>
> **/QIfist** is only available in the compiler targeting x86; this compiler option is not available in the compilers targeting x64 orARM.

In addition to converting from a floating-point type to integral type, the `_ftol` function ensures the rounding mode of the floating-point unit (FPU) is toward zero (truncate), by setting bits 10 and 11 of the control word. This guarantees that converting from a floating-point type to an integral type occurs as described by the ANSI C standard (the fractional portion of the number is discarded). When using **/QIfist**, this guarantee no longer applies. The rounding mode will be one of four as documented in Intel reference manuals:

- Round toward nearest (even number if equidistant)
- Round toward negative infinity
- Round toward positive infinity
- Round toward zero

You can use the _control87, _controlfp, __control87_2 C Run-Time function to modify the rounding behavior of the FPU. The default rounding mode of the FPU is "Round toward nearest." Using **/QIfist** can improve the performance of your application, but not without risk. You should thoroughly test the portions of your code that are sensitive to

rounding modes before relying upon code built with **/QIfist** in production environments.

/arch (x86) and **/QIfist** can not be used on the same compiland.

> ⓘ **Note**
>
> **/QIfist** is not in effect by default because the rounding bits also affect floating point to floating point rounding (which occurs after every calculation), so when you set the flags for C-style (toward zero) rounding, your floating point calculations might be different. **/QIfist** should not be used if your code depends upon the expected behavior of truncating the fractional portion of the floating-point number. If you are unsure, do not use **/QIfist**.

The **/QIfist** option is deprecated starting in Visual Studio 2005. The compiler has made significant improvements in float to int conversion speed. For a list of deprecated compiler options, see **Deprecated and Removed Compiler Options** in Compiler Options Listed by Category.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Type the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/Q Options (Low-Level Operations)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Qimprecise_fwaits (Remove fwaits Inside Try Blocks)

Article • 08/03/2021

Removes the `fwait` commands internal to `try` blocks when you use the /fp:except compiler option.

Syntax

```
/Qimprecise_fwaits
```

Remarks

This option has no effect if `/fp:except` isn't also specified. If you specify the `/fp:except` option, the compiler will insert a `fwait` instruction around each line of code in a `try` block. In this way, the compiler can identify the specific line of code that produces an exception. `/Qimprecise_fwaits` removes internal `fwait` instructions, leaving only the waits around the `try` block. It improves performance, but the compiler can only show which `try` block causes an exception, not which line.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/Q Options (Low-Level Operations)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/QIntel-jcc-erratum

Article • 08/03/2021

Specifies that the compiler generates instructions to mitigate the performance impact caused by the Intel Jump Conditional Code (JCC) erratum microcode update in certain Intel processors.

Syntax

> **/QIntel-jcc-erratum**

Remarks

Under **/QIntel-jcc-erratum**, the compiler detects jump and macro-fused jump instructions that cross or end on a 32-byte boundary. It aligns these instructions to the boundary. This change mitigates the performance impact of microcode updates that prevent the JCC erratum in certain Intel processors. For more information about the erratum, see Mitigations for Jump Conditional Code Erratum on the Intel website.

The **/QIntel-jcc-erratum** option is available in Visual Studio 2019 version 16.5 and later. This option is only available in compilers that target x86 and x64. The option isn't available in compilers that target ARM processors.

The **/QIntel-jcc-erratum** option is off by default, and works only in optimized builds. This option can increase code size.

/QIntel-jcc-erratum is incompatible with /clr.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Select a value for the **Enable Intel JCC Erratum Mitigation** property. Choose **OK** to apply the change.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/Q options (Low-level operations)
MSVC compiler options
MSVC compiler command-line syntax

`/Qpar` (Auto-parallelizer)

Article • 02/23/2022

Enables the Auto-parallelizer feature of the compiler to automatically parallelize loops in your code.

Syntax

> `/Qpar`

Remarks

When the compiler automatically parallelizes loops in code, it spreads computation across multiple processor cores. The compiler parallelizes a loop only if it determines that it's legal to do so and that parallelization would improve performance.

The `#pragma loop()` directives are available to help the optimizer parallelize specific loops. For more information, see loop.

For information about how to enable output messages for the auto-parallelizer, see /Qpar-report (Auto-parallelizer reporting level).

To set the `/Qpar` compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify the **Enable Parallel Code Generation** property. Choose **OK** or **Apply** to save your changes.

To set the `/Qpar` compiler option programmatically

- Use the code example in AdditionalOptions.

See also

/Q options (Low-level operations)
/Qpar-report (Auto-parallelizer reporting level)
MSVC compiler options
MSVC compiler command-line syntax
#pragma loop()
Native code vectorization in Visual Studio

/Qpar-report (Auto-Parallelizer Reporting Level)

Article • 08/03/2021

Enables the reporting feature of the compiler's Auto-Parallelizer and specifies the level of informational messages for output during compilation.

Syntax

```
/Qpar-report:{1}{2}
```

Remarks

/Qpar-report:1
Outputs an informational message for loops that are parallelized.

/Qpar-report:2
Outputs an informational message for loops that are parallelized and also for loops that are not parallelized, together with a reason code.

Messages are reported to stdout. If no informational messages are reported, then either the code contains no loops, or the reporting level was not set to report loops that are not parallelized. For more information about reason codes and messages, see Vectorizer and Parallelizer Messages.

To set the /Qpar-report compiler option in Visual Studio

1. In **Solution Explorer**, open the shortcut menu for the project and then choose **Properties**.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In the **Additional Options** box, enter `/Qpar-report:1` or `/Qpar-report:2`.

To set the /Qpar-report compiler option programmatically

- Use the code example in AdditionalOptions.

See also

/Q Options (Low-Level Operations)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Native code vectorization in Visual Studio

/Qsafe_fp_loads

Article • 08/03/2021

Requires integer move instructions for floating-point values and disables certain floating-point load optimizations.

Syntax

> **/Qsafe_fp_loads**

Remarks

/Qsafe_fp_loads is only available in the compilers that target x86; it is not available in the compilers that target x64 or ARM.

/Qsafe_fp_loads forces the compiler to use integer move instructions instead of floating-point move instructions to move data between memory and MMX registers. This option also disables register load optimization for floating-point values that can be loaded in multiple control paths when the value may cause an exception on load—for example, a NaN value.

This option is overridden by /fp:except. **/Qsafe_fp_loads** specifies a subset of the compiler behavior that's specified by **/fp:except**.

/Qsafe_fp_loads is incompatible with /clr and /fp:fast. For more information about floating point compiler options, see /fp (Specify Floating-Point Behavior).

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box. Choose **OK** to apply the change.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/Q Options (Low-Level Operations)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Qspectre

Article • 11/04/2021

Specifies compiler generation of instructions to mitigate certain Spectre variant 1 security vulnerabilities.

Syntax

> `/Qspectre`

Remarks

The `/Qspectre` option causes the compiler to insert instructions to mitigate certain Spectre security vulnerabilities. These vulnerabilities are called *speculative execution side-channel attacks*. They affect many operating systems and modern processors, including processors from Intel, AMD, and ARM.

The `/Qspectre` option is available starting in Visual Studio 2017 version 15.5.5 and all later versions. It's available in Visual Studio 2015 Update 3 through KB 4338871.

The `/Qspectre` option is off by default.

In its initial release, the `/Qspectre` option only worked on optimized code. Starting in Visual Studio 2017 version 15.7, the `/Qspectre` option is supported at all optimization levels.

Several Microsoft C++ libraries are also available in versions with Spectre mitigation. The Spectre-mitigated libraries for Visual Studio can be downloaded in the Visual Studio Installer. They're found in the **Individual Components** tab under **Compilers, build tools, and runtimes**, and have "Libs for Spectre" in the name. Both DLL and static runtime libraries with mitigation enabled are available for a subset of the Visual C++ runtimes: VC++ start-up code, vcruntime140, msvcp140, concrt140, and vcamp140. The DLLs are supported for application-local deployment only. The contents of the Visual C++ Runtime Libraries Redistributable haven't been modified.

You can also install Spectre-mitigated libraries for MFC and ATL. They're found in the **Individual Components** tab under **SDKs, libraries, and frameworks**.

> **Note**

There are no versions of Spectre-mitigated libraries for Universal Windows (UWP) apps or components. App-local deployment of such libraries isn't possible.

Applicability

If your code operates on data that crosses a trust boundary, then we recommend you use the `/Qspectre` option to rebuild and redeploy your code to mitigate this issue as soon as possible. An example of such code is code that loads untrusted input that can affect execution. For example, code that makes remote procedure calls, parses untrusted input or files, or uses other local inter-process communication (IPC) interfaces. Standard sandboxing techniques may not be sufficient. Investigate your sandboxes carefully before you decide your code doesn't cross a trust boundary.

Availability

The `/Qspectre` option is available starting in Visual Studio 2017 version 15.5.5, and in all updates to Microsoft C/C++ compilers (MSVC) made on or after January 23, 2018. Use the Visual Studio Installer to update the compiler, and to install the Spectre-mitigated libraries as individual components. The `/Qspectre` option is also available in Visual Studio 2015 Update 3 through a patch. For more information, see KB 4338871.

All versions of Visual Studio 2017 version 15.5, and all Previews of Visual Studio 2017 version 15.6. include an undocumented option, `/d2guardspecload`. It's equivalent to the initial behavior of `/Qspectre`. You can use `/d2guardspecload` to apply the same mitigations to your code in these versions of the compiler. We recommend you update your build to use `/Qspectre` in compilers that support the option. The `/Qspectre` option may also support new mitigations in later versions of the compiler.

Effect

The `/Qspectre` option outputs code to mitigate Specter variant 1, Bounds Check Bypass, CVE-2017-5753. It works by insertion of instructions that act as a speculative code execution barrier. The specific instructions used to mitigate processor speculation depend upon the processor and its micro-architecture, and may change in future versions of the compiler.

When you enable the `/Qspectre` option, the compiler attempts to identify instances where speculative execution may bypass bounds checks. That's where it inserts the barrier instructions. It's important to be aware of the limits to the analysis that a

compiler can do to identify instances of variant 1. As such, there's no guarantee that all possible instances of variant 1 are instrumented under `/Qspectre`.

Performance impact

The effect of `/Qspectre` on performance appeared to be negligible in several sizable code bases. However, there are no guarantees that performance of your code under `/Qspectre` remains unaffected. You should benchmark your code to determine the effect of the option on performance. If you know that the mitigation isn't required in a performance-critical block or loop, you can selectively disable the mitigation by use of a __declspec(spectre(nomitigation)) directive. This directive isn't available in compilers that only support the `/d2guardspecload` option.

Required libraries

The `/Qspectre` compiler option mitigates issues in your own code. For greater protection, we strongly recommend you also use libraries built to provide Spectre mitigations. Several of the Microsoft runtime libraries are available with Spectre mitigations.

These libraries are optional components that must be installed by using the Visual Studio Installer:

- MSVC version *version_numbers* Libs for Spectre [(x86 and x64) | (ARM) | (ARM64)]
- Visual C++ ATL for [(x86/x64) | ARM | ARM64] with Spectre Mitigations
- Visual C++ MFC for [x86/x64 | ARM | ARM64] with Spectre Mitigations

The default MSBuild-based project system in the Visual Studio IDE lets you specify a Spectre Mitigation property for your projects. This property sets the `/Qspectre` compiler option and changes the library paths to link the Spectre-mitigated runtime libraries. If these libraries aren't installed when you build your code, the build system reports warning MSB8040. If your MFC or ATL code fails to build, and the linker reports an error such as "fatal error LNK1104: cannot open file 'oldnames.lib'", these missing libraries may be the cause.

There are several ways to specify the Spectre-mitigated libraries to the build command line. You can specify the path to the Spectre-mitigated libraries by using the /LIBPATH linker option to make them the default libraries. You can use the /NODEFAULTLIB linker option and explicitly link the Spectre-mitigated libraries. Or, you can set the `LIBPATH` environment variable to include the path to the Spectre-mitigated libraries for your target platform. One way to set this path in the environment is to use a developer

command prompt set up by using the `spectre_mode` option. For more information, see Use the developer tools in an existing command window.

Additional information

For more information, see the official Microsoft Security Advisory ADV180002, Guidance to mitigate speculative execution side-channel vulnerabilities. Guidance is also available from Intel, Speculative Execution Side Channel Mitigations, and ARM, Cache Speculation Side-channels.

For a Windows-specific overview of Spectre and Meltdown mitigations, see Understanding the performance impact of Spectre and Meltdown mitigations on Windows Systems.

For an overview of Spectre vulnerabilities addressed by the MSVC mitigations, see Spectre mitigations in MSVC on the C++ Team Blog.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Select a new value for the **Spectre Mitigation** property. Choose **OK** to apply the change.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/Q options (Low-level operations)
MSVC compiler options
MSVC compiler command-line syntax

`/Qspectre-jmp`

Article • 12/01/2023

Causes the compiler to generate an `int3` instruction (software interrupt) after unconditional direct branches. This option extends the /Qspectre flag and mitigates speculative execution side-channel attacks on unconditional direct branches.

Syntax

> **/Qspectre-jmp**

Remarks

`/Qspectre-jmp` causes the compiler to detect executable instructions following unconditional direct branches. An `int3` is inserted following unconditional direct branches to ensure that no instructions are speculatively executed beyond the branch. For example, the compiler mitigates `jmp addr` by adding an `int3` instruction following the `jmp` instruction as shown here:

```asm
jmp addr
int3
```

`/Qspectre-jmp` is off by default. It's supported for all optimization levels.

Set this compiler option programmatically

To set this option programmatically, see VCCLCompilerTool.AdditionalOptions property.

See also

/Qspectre
/Qspectre-jmp
/Qspectre-load
/Qspectre-load-cf
/Q options (Low-Level Operations)

MSVC compiler options

MSVC compiler command-line syntax

/Qspectre-load

Article • 12/01/2023

Specifies compiler generation of serializing instructions for every load instruction. This option extends the **/Qspectre** flag, mitigating against any possible **speculative execution side-channel attacks** based on loads.

Syntax

> **/Qspectre-load**

Remarks

/Qspectre-load causes the compiler to detect loads from memory, and insert serializing instructions after them. Control flow instructions that load memory, including `RET` and `CALL`, are split into a load and a control flow transfer. The load is followed by an `LFENCE` to ensure the load is protected. There are cases where the compiler can't split control flow instructions, such as the `jmp` instruction, so it uses an alternate mitigation technique. For example, the compiler mitigates `jmp [rax]` by adding instructions to load the target nondestructively before inserting an LFENCE, as shown here:

asm

```
    xor rbx, [rax]
    xor rbx, [rax]  ; force a load of [rax]
    lfence          ; followed by an LFENCE
    jmp [rax]
```

Because **/Qspectre-load** stops speculation of all loads, the performance impact is high. The mitigation isn't appropriate everywhere. If there are performance critical blocks of code that don't require protection, you can disable these mitigations by using `__declspec(spectre(nomitigation))`. For more information, see __declspec spectre.

The **/Qspectre-load** option is off by default, and supports all optimization levels.

The **/Qspectre-load** option is available in Visual Studio 2019 version 16.5 and later. This option is only available in compilers that target x86 and x64 processors. It's not available in compilers that target ARM processors.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Select a new value for the **Spectre Mitigation** property. Choose **OK** to apply the change.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/Qspectre
/Qspectre-jmp
/Qspectre-load-cf
/Q options (Low-Level Operations)
MSVC compiler options
MSVC compiler command-line syntax

/Qspectre-load-cf

Article • 12/01/2023

Specifies compiler generation of serializing instructions for every control-flow instruction that contains a load. This option performs a subset of the mitigations done by the /Qspectre-load option.

Syntax

> **/Qspectre-load-cf**

Remarks

/Qspectre-load-cf causes the compiler to detect `JMP`, `RET`, and `CALL` control-flow instructions that load from memory, and to insert serializing instructions after the load. Where possible, these instructions are split into a load and a control flow transfer. The load is followed by an `LFENCE` to ensure the load is protected. There are cases where the compiler can't split instructions, such as the `JMP` instruction, so it uses an alternate mitigation technique. For example, the compiler mitigates `jmp [rax]` by adding instructions to load the target nondestructively before inserting an LFENCE, as shown here:

asm

```
    xor rbx, [rax]
    xor rbx, [rax]  ; force a load of [rax]
    lfence          ; followed by an LFENCE
    jmp [rax]
```

Because **/Qspectre-load-cf** stops speculation of all loads in control-flow instructions, the performance impact is high. The mitigation isn't appropriate everywhere. If there are performance critical blocks of code that don't require protection, you can disable these mitigations by using `__declspec(spectre(nomitigation))`.

The **/Qspectre-load-cf** option is off by default, and supports all optimization levels.

The **/Qspectre-load-cf** option is available in Visual Studio 2019 version 16.5 and later. This option is only available in compilers that target x86 and x64 processors. It's not available in compilers that target ARM processors.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Select a new value for the **Spectre Mitigation** property. Choose **OK** to apply the change.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/Qspectre
/Qspectre-jmp
/Qspectre-load
/Q options (Low-level operations)
MSVC compiler options
MSVC compiler command-line syntax

/Qvec-report (Auto-Vectorizer Reporting Level)

Article • 08/03/2021

Enables the reporting feature of the compiler Auto-Vectorizer and specifies the level of informational messages for output during compilation.

Syntax

```
/Qvec-report:{1}{2}
```

Remarks

/Qvec-report:1
Outputs an informational message for loops that are vectorized.

/Qvec-report:2
Outputs an informational message for loops that are vectorized and for loops that are not vectorized, together with a reason code.

For information about reason codes and messages, see Vectorizer and Parallelizer Messages.

To set the /Qvec-report compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. In the **Additional Options** box, enter `/Qvec-report:1` or `/Qvec-report:2`.

To set the /Qvec-report compiler option programmatically

- Use the code example in AdditionalOptions.

See also

/Q Options (Low-Level Operations)
MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Native code vectorization in Visual Studio

`/RTC` (Run-time error checks)

Article • 08/03/2021

Used to enable and disable the run-time error checks feature, in conjunction with the runtime_checks pragma.

Syntax

> `/RTC1`
>
> `/RTCc`
>
> `/RTCs`
>
> `/RTCu`

Arguments

`/RTC1`

Equivalent to `/RTCsu`.

`/RTCc`

Reports when a value is assigned to a smaller data type and results in a data loss. For example, it reports if a `short` type value of `0x0101` is assigned to a variable of type `char`.

This option can report situations in which you intend to truncate. For example, when you want the first 8 bits of an `int` returned as a `char`. Because `/RTCc` causes a run-time error if an assignment causes any information loss, first mask off the information you need to avoid the run-time error. For example:

C

```
#include <crtdbg.h>

char get8bits(unsigned value, int position) {
   _ASSERT(position < 32);
   return (char)(value >> position);
   // Try the following line instead:
   // return (char)((value >> position) & 0xff);
}

int main() {
   get8bits(12341235,3);
}
```

Because `/RTCc` rejects code that conforms to the standard, it's not supported by the C++ Standard Library. Code that uses `/RTCc` and the C++ Standard Library may cause compiler error C1189. You can define `_ALLOW_RTCc_IN_STL` to silence the warning and use the `/RTCc` option.

`/RTCs`

Enables stack frame run-time error checking, as follows:

- Initialization of local variables to a nonzero value. This option helps identify bugs that don't appear when running in debug mode. There's a greater chance that stack variables still have a zero value in a debug build compared to a release build. That's because of compiler optimizations of stack variables in a release build. Once a program has used an area of its stack, it's never reset to 0 by the compiler. That means any uninitialized stack variables that happen to use the same stack area later can return values left over from the earlier use of this stack memory.
- Detection of overruns and underruns of local variables such as arrays. `/RTCs` doesn't detect overruns when accessing memory that results from compiler padding within a structure. Padding could occur by using align, /Zp (Struct Member Alignment), or pack, or if you order structure elements in such a way as to require the compiler to add padding.
- Stack pointer verification, which detects stack pointer corruption. Stack pointer corruption can be caused by a calling convention mismatch. For example, using a function pointer, you call a function in a DLL that is exported as __stdcall but you declare the pointer to the function as __cdecl.

`/RTCu`

Reports when a variable is used without having been initialized. For example, an instruction that generates warning C4701 may also generate a run-time error under `/RTCu`. Any instruction that generates Compiler Warning (level 1 and level 4) C4700 will generate a run-time error under `/RTCu`.

However, consider the following code fragment:

C++

```
int a, *b, c;
if ( 1 )
b = &a;
c = a;  // No run-time error with /RTCu
```

If a variable could have been initialized, it's not reported at run time by `/RTCu`. For example, after a variable is aliased through a pointer, the compiler doesn't track the variable and report uninitialized uses. In effect, you can initialize a variable by taking its address. The `&` operator works like an assignment operator in this situation.

Remarks

Run-time error checks are a way for you to find problems in your running code; for more information, see How to: Use native run-time checks.

You can specify more than one `/RTC` option on the command line. The option arguments may be combined; for example, `/RTCcu` is the same as `/RTCc /RTCu`.

If you compile your program at the command line using any of the `/RTC` compiler options, any pragma optimize instructions in your code silently fail. That's because run-time error checks aren't valid in a release (optimized) build.

Use `/RTC` for development builds; Don't use `/RTC` for a release build. `/RTC` can't be used with compiler optimizations (/O Options (Optimize Code)). A program image built with `/RTC` is slightly larger and slightly slower than an image built with `/Od` (up to 5 percent slower than an `/Od` build).

The `__MSVC_RUNTIME_CHECKS` preprocessor directive will be defined when you use any `/RTC` option or /GZ.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify one or both of the following properties: **Basic Runtime Checks** or **Smaller Type Check**.

To set this compiler option programmatically

- See BasicRuntimeChecks and SmallerTypeCheck properties.

See also

MSVC compiler options

MSVC compiler command-line syntax

How to: Use native run-time checks

`/scanDependencies` (List module dependencies in standard form)

Article • 03/02/2023

This compiler option generates a JSON file that lists module and header-unit dependencies according to C++ Standard proposal P1689R5 Format for describing dependencies of source files.

Syntax

> **`/scanDependencies-`**
>
> **`/scanDependencies`** *`filename`*
>
> **`/scanDependencies`** *`directory`*

Arguments

`-`

If the single dash is provided, then the compiler emits the source dependencies JSON to `stdout`, or to where compiler output is redirected.

`filename`

The compiler writes the source dependency output to the specified filename, which may include a relative or absolute path. The file is created if it doesn't exist.

`directory`

If the argument is a directory, the compiler generates source dependency files in the specified directory. The directory must exist, or the argument is treated as a *`filename`*. The output file name is based on the full name of the input file, with an appended *`.module.json`* extension. For example, if the file provided to the compiler is *`main.cpp`*, the generated output filename is *`main.cpp.module.json`*.

Remarks

The **`/scanDependencies`** compiler option identifies which dependencies, modules, and header units must be compiled before you can compile the project that uses them. For instance, it lists `import <library>;` or `import "library";` as a header unit dependency, and `import name;` as a module dependency. The intent is to provide this information in

a common format consumable by build tools such as CMake. To report module and header unit dependencies, you must also compile by using /std:c++20 or later.

This command-line option is similar to /sourceDependencies:directives and /sourceDependencies, but differs in the following ways:

- The output uses the P1689R5 schema, instead of the Microsoft-specific schema generated by `/sourceDependencies:directives`.
- Unlike `/sourceDependencies`, the compiler doesn't produce compiled output. Instead, the files are scanned for module directives. No compiled code, modules, or header units are produced.
- The output JSON file doesn't list imported modules and imported header units (*`.ifc`* files) because this option only scans the project files. There are no built modules or header units to list.
- Only directly imported modules or header units are listed. It doesn't list the dependencies of the imported modules or header units themselves.
- Textually included header files such as `#include <file>` or `#include "file"` aren't listed as dependencies unless translated to a header unit by using the /translateInclude option.
- `/scanDependencies` is meant to be used before *`.ifc`* files are built.

`/scanDependencies` is available starting in Visual Studio 2022 version 17.2. It's not enabled by default.

When you specify the /MP (Build with multiple processes) compiler option, we recommend that you use `/scanDependencies` with a directory argument. If you provide a single filename argument, two instances of the compiler may attempt to open the output file simultaneously and cause an error. Use of `/MP` with `/scanDependencies-` to send output to `stdout` could cause interleaved results.

When a non-fatal compiler error occurs, the dependency information still gets written to the output file.

All file paths appear as absolute paths in the output.

For details on the format and schema used in the output JSON file, see P1689R5 section 6.

Examples

Consider the following sample code:

```
C++
```

```
//app.cpp:
#include <vector>

import other.module;
import std.core;

import "t.h";

import <iostream>;

int main() {}
```

You can use this command line to report dependencies in *app.cpp*:

```
cl /std:c++latest /scanDependencies output.json app.cpp
```

The compiler produces a JSON file, *output.json*, with content similar to:

JSON

```
{
    "version": 1,
    "revision": 0,
    "rules": [
        {
            "primary-output": "app.obj",
            "outputs": [
                "C:\\Users\\username\\source\\repos\\app\\app"
            ],
            "requires": [
                {
                    "logical-name": "other.module"
                },
                {
                    "logical-name": "std.core"
                },
                {
                    "logical-name": "t.h",
                    "source-path":
"C:\\Users\\username\\source\\repos\\app\\app\\t.h",
                    "lookup-method": "include-quote",
                    "unique-on-source-path": true
                },
                {
                    "logical-name": "iostream",
                    "source-path": "C:\\Program
Files\\...\\include\\iostream",
                    "lookup-method": "include-angle",
                    "unique-on-source-path": true
                }
            ]
```

```
        }
    ]
}
```

We've used `...` to abbreviate the reported paths. The report contains the absolute paths. The paths reported depend on where the compiler finds the dependencies. If the results are unexpected, you may want to check your project's include path settings.

No `.ifc` files are listed in the output because they weren't built. Unlike `/sourceDependencies`, the compiler doesn't produce compiled output when `/scanDependencies` is specified, so no compiled modules or header units are produced to import.

To set this compiler option in Visual Studio

You normally shouldn't set the `/scanDependencies` option in the Visual Studio development environment. The compiler doesn't generate object files when you set this option, which makes the link step fail and report an error.

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to add `/scanDependencies-` or `/scanDependencies "pathname"`, where `"pathname"` refers to a directory for output.
4. Choose **OK** to save your changes.

To report module and header unit dependencies, you must also set the **Configuration Properties** > **General** > **C++ Language Standard** property to **ISO C++20 Standard** or later.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax
/sourceDependencies:directives

/sourceDependencies
/std (Specify language standard version)
/translateInclude

/sdl (Enable Additional Security Checks)

Article • 08/03/2021

Enables recommended Security Development Lifecycle (SDL) checks. These checks change security-relevant warnings into errors, and set additional secure code-generation features.

Syntax

> `/sdl`[`-`]

Remarks

/sdl enables a superset of the baseline security checks provided by /GS and overrides `/GS-`. By default, `/sdl` is off. `/sdl-` disables the additional security checks.

Compile-time Checks

`/sdl` enables these warnings as errors:

Warning enabled by /sdl	Equivalent command-line switch	Description
C4146	/we4146	A unary minus operator was applied to an unsigned type, resulting in an unsigned result.
C4308	/we4308	A negative integral constant converted to unsigned type, resulting in a possibly meaningless result.
C4532	/we4532	Use of `continue`, `break`, or `goto` keywords in a `__finally`/`finally` block has undefined behavior during abnormal termination.
C4533	/we4533	Code initializing a variable will not be executed.
C4700	/we4700	Use of an uninitialized local variable.
C4703	/we4703	Use of a potentially uninitialized local pointer variable.
C4789	/we4789	Buffer overrun when specific C run-time (CRT) functions are used.
C4995	/we4995	Use of a function marked with pragma deprecated.

Warning enabled by /sdl	Equivalent command-line switch	Description
C4996	/we4996	Use of a function marked as deprecated.

Runtime checks

When `/sdl` is enabled, the compiler generates code that does these checks at run time:

- Enables the strict mode of `/GS` run-time buffer overrun detection, equivalent to compiling with `#pragma strict_gs_check(push, on)`.
- Does limited pointer sanitization. In expressions that don't involve dereferences and in types that have no user-defined destructor, pointer references are set to a non-valid address after a call to `delete`. This sanitization helps to prevent the reuse of stale pointer references.
- Initializes class member pointers. Automatically initializes class members of pointer type to `nullptr` on object instantiation (before the constructor runs). It helps prevent the use of uninitialized pointers that the constructor doesn't explicitly initialize. The compiler-generated member pointer initialization is called as long as:
 - The object isn't allocated using a custom (user defined) `operator new`
 - The object isn't allocated as part of an array (for example `new A[x]`)
 - The class isn't managed or imported
 - The class has a user-defined default constructor.

 To be initialized by the compiler-generated class initialization function, a member must be a pointer, and not a property or constant.

For more information, see Warnings, /sdl, and improving uninitialized variable detection.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **General** property page.

3. Set the **SDL checks** property by using the property drop-down control. Choose **OK** or **Apply** to save your changes.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/showIncludes` (List include files)

Article • 05/25/2022

Causes the compiler to output a list of the include files. The option also displays nested include files, that is, the files included by the files that you include.

Syntax

> `/showIncludes`

Remarks

When the compiler comes to an include file during compilation, a message is output, as in this example:

Windows Command Prompt

```
Note: including file: d:\MyDir\include\stdio.h
```

Nested include files are indicated by an indentation, one space for each level of nesting, as in this example:

Windows Command Prompt

```
Note: including file: d:\temp\1.h
Note: including file:  d:\temp\2.h
```

In this case, `2.h` was included from within `1.h`, causing the indentation.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Advanced** property page.
3. Modify the **Show Includes** property.

To set this compiler option programmatically

- See ShowIncludes.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/source-charset` (Set source character set)

Article • 02/01/2022

This option lets you specify the source character set for your executable.

Syntax

> **`/source-charset:`**[*`IANA_name`* | *`.CPID`*]

Arguments

`IANA_name`
The IANA-defined character set name.

`.CPID`
The code page identifier as a decimal number, preceded by a `.` character.

Remarks

You can use the **`/source-charset`** option to specify an extended source character set to use when your source files include characters that are not represented in the basic source character set. The source character set is the encoding used to interpret the source text of your program. It's converted into the internal representation used as input to the preprocessing phases before compilation. The internal representation is then converted to the execution character set to store string and character values in the executable. You can use either the IANA or ISO character set name, or a dot (`.`) followed by 3-5 decimal digits that specify the code page identifier of the character set to use. For a list of supported code page identifiers and character set names, see Code Page Identifiers.

By default, Visual Studio detects a byte-order mark to determine if the source file is in an encoded Unicode format, for example, UTF-16 or UTF-8. If no byte-order mark is found, it assumes that the source file is encoded in the current user code page, unless you use the **`/source-charset`** or **`/utf-8`** option to specify a character set name or code page. Visual Studio allows you to save your C++ source code in any of several character encodings. For more information about source and execution character sets, see Character sets in the language documentation.

The source character set you supply must map the 7-bit ASCII characters to the same code points in your character set, or many compilation errors are likely to follow. Your source character set must also have a mapping to the extended Unicode character set of UTF-8. Characters that have no equivalent in UTF-8 are represented by an implementation-specific substitute. The Microsoft compiler uses a question mark for these characters.

If you want to set both the source character set and the execution character set to UTF-8, you can use the `/utf-8` compiler option as a shortcut. It's equivalent to `/source-charset:utf-8 /execution-charset:utf-8` on the command line. Any of these options also enables the `/validate-charset` option by default.

To set this compiler option in the Visual Studio development environment

1. Open the **Property Pages** dialog box for your project. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In **Additional Options**, add the `/source-charset` option, and specify your preferred encoding.
4. Choose **OK** to save your changes.

See also

MSVC compiler options
MSVC compiler command-line syntax
/execution-charset (Set execution character set)
/utf-8 (Set source and execution character sets to UTF-8)
/validate-charset (Validate for compatible characters)

`/sourceDependencies` (List all source-level dependencies)

Article • 02/23/2024

This command-line switch generates a JSON file that details the source-level dependencies consumed during compilation. The JSON file contains a list of the source dependencies, which include:

- Header files. Both directly included and the list of headers included by those headers.
- The PCH used (if `/Yu` is specified).
- Names of imported modules
- File paths and names of both directly imported header units and of the modules and header units they import in turn.

This option provides information necessary to build modules and header units in the proper dependency order.

Syntax

> **`/sourceDependencies-`**
>
> **`/sourceDependencies`** *filename*
>
> **`/sourceDependencies`** *directory*

Arguments

`-`

If the single dash is provided, then the compiler will emit the source dependencies JSON to `stdout`, or to where compiler output is redirected.

`filename`

The compiler writes the source dependency output to the specified filename, which may include a relative or absolute path. The file is created if it doesn't exist.

`directory`

If the argument is a directory, the compiler generates source dependency files in the specified directory. The directory must exist, or the argument is treated as a *`filename`*. The output file name is based on the full name of the input file, with an appended *`.json`*

extension. For example, if the file provided to the compiler is `main.cpp`, the generated output filename is `main.cpp.json`.

Remarks

The **`/sourceDependencies`** compiler option is available starting in Visual Studio 2019 version 16.7. It's not enabled by default.

When you specify the /MP (Build with multiple processes) compiler option, we recommend you use **`/sourceDependencies`** with a directory argument. If you provide a single filename argument, two instances of the compiler may attempt to open the output file simultaneously and cause an error. Use of **`/MP`** with **`/sourceDependencies-`** to send output to `stdout` could cause interleaved results.

When a non-fatal compiler error occurs, the dependency information still gets written to the output file.

All file paths appear as absolute paths in the output.

Examples

Given the following sample code:

C++

```
// ModuleE.ixx:
export module ModuleE;
import ModuleC;
import ModuleD;
import <iostream>;
```

You can use **`/sourceDependencies`** with the rest of your compiler options:

```
cl ... /sourceDependencies output.json ... main.cpp
```

where `...` represents your other compiler options. This command line produces a JSON file `output.json` with content like:

JSON

```
{
    "Version": "1.2",
    "Data": {
```

```
        "Source": "F:\\Sample\\myproject\\modulee.ixx",
        "ProvidedModule": "ModuleE",
        "Includes": [],
        "ImportedModules": [
            {
                "Name": "ModuleC",
                "BMI":
"F:\\Sample\\Outputs\\Intermediate\\MyProject\\x64\\Debug\\ModuleC.ixx.ifc"
            },
            {
                "Name": "ModuleB",
                "BMI":
"F:\\Sample\\Outputs\\Intermediate\\ModuleB\\x64\\Debug\\ModuleB.ixx.ifc"
            },
            {
                "Name": "ModuleD",
                "BMI":
"F:\\Sample\\Outputs\\Intermediate\\MyProject\\x64\\Debug\\ModuleD.cppm.ifc"
            }
        ],
        "ImportedHeaderUnits": [
            {
                "Header": "f:\\visual studio 16
main\\vc\\tools\\msvc\\14.29.30030\\include\\iostream",
                "BMI":
"F:\\Sample\\Outputs\\Intermediate\\HeaderUnits\\x64\\Debug\\iostream_W4L4JY
GFJ3GL8OG9.ifc"
            }
        ]
    }
}
```

We've used `...` to abbreviate the reported paths. The report contains the absolute paths. The paths reported depend on where the compiler finds the dependencies. If the results are unexpected, you may want to check your project's include path settings.

`ProvidedModule` lists exported module or module partition names.

To set this compiler option in the Visual Studio development environment

You normally shouldn't set this option yourself in the Visual Studio development environment. It's set by the build system.

See also

MSVC compiler options

MSVC compiler command-line syntax

/scanDependencies
/sourceDependencies:directives

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/sourceDependencies:directives` (List module and header unit dependencies)

Article • 05/20/2022

This command-line option scans source files and their `#include` statements to generate a JSON file that lists module export and imports. This information can be used by a build system to determine the build order of modules and header units.

This option differs from /sourceDependencies in the following ways:

- The compiler doesn't produce compiled output. No compiled code, modules, or header units are produced. Instead, the files are scanned for module directives.
- The JSON format is different from what `/sourceDependencies` produces. The `/sourceDependencies` option is intended to be used with other build tools, such as CMake.
- The output JSON file doesn't list imported modules and imported header units (*`.ifc`* files) because this option does a scan of the project files, not a compilation. So there are no built modules or header units to list.
- Only directly imported modules or header units are listed. It doesn't list the dependencies of the imported modules or header units themselves.
- Header file dependencies aren't listed. That is, `#include <file>` or `#include "file"` dependencies aren't listed.
- `/sourceDependencies:directives` is meant to be used before *`.ifc`* files are built.
- `/sourceDependencies` causes the compiler to report all of the files, such as `#includes`, `.pch` files, `.ifc` files, and so on, that were used for a particular translation unit, whereas `/sourceDependencies:directives [file1]` scans the specified source file and reports all `import` and `export` statements. `/sourceDependencies` can be used with `/sourceDependencies:directives`.

Syntax

> **`/sourceDependencies:directives-`**
>
> **`/sourceDependencies:directives`** *filename*
>
> **`/sourceDependencies:directives`** *directory*

Arguments

`-`

If the single dash is provided, then the compiler will emit the source dependencies JSON to `stdout`, or to where compiler output is redirected.

`filename`

The compiler writes the source dependency output to the specified filename, which may include a relative or absolute path. The file is created if it doesn't exist.

`directory`

If the argument is a directory, the compiler generates source dependency files in the specified directory. The directory must exist, or the argument is treated as a *`filename`*. The output file name is based on the full name of the input file, with an appended *`.json`* extension. For example, if the file provided to the compiler is *`main.cpp`*, the generated output filename is *`main.cpp.json`*.

Remarks

`/sourceDependencies:directives` is available starting in Visual Studio 2019 version 16.10.

When you specify the /MP (Build with multiple processes) compiler option, we recommend that you use **`/sourceDependencies:directives`** with a directory argument. This option makes the compiler output a separate *`*.module.json`* file for each source file. If you provide a single filename argument, two instances of the compiler may attempt to open the output file simultaneously and cause an error. Use of **`/MP`** with **`/sourceDependencies:directives-`** to send output to `stdout` could cause interleaved results.

When a non-fatal compiler error occurs, the dependency information still gets written to the output file.

All file paths appear as absolute paths in the output.

This switch can be used with /translateInclude.

Examples

Given the following sample code:

C++

```
//main.cpp:
#include <vector>
```

```
import m;
import std.core;

import <utility>;

import "t.h";

int main() {}
```

This following command line:

`cl /std:c++latest /translateInclude /sourceDependencies:directives output.json main.cpp`

produces a JSON file *`output.json`* similar to:

JSON

```
{
   "Version":"1.1",
   "Data":{
      "Source":"C:\\a\\b\\main.cpp",
      "ProvidedModule":"",
      "ImportedModules":[
         "m",
         "std.core"
      ],
      "ImportedHeaderUnits":[
         "C:\\...\\utility",
         "C:\\a\\b\\t.h"
      ]
   }
}
```

For brevity, the previous example uses `...` to abbreviate the reported paths. The report contains the absolute paths. The paths reported depend on where the compiler finds the dependencies. If the results are unexpected, you might want to check your project's include path settings.

`ProvidedModule` lists exported module or module partition names.

No *`.ifc`* files are listed in the output because they weren't built. Unlike `/sourceDependencies`, the compiler doesn't produce compiled output when `/sourceDependencies:directives` is specified, so no compiled modules or header units are produced.

To set this compiler option in Visual Studio

You normally shouldn't set this option yourself in the Visual Studio development environment. It's set by the build system.

See also

/translateInclude
C++ header-units.json reference
MSVC compiler options
MSVC compiler command-line syntax
/scanDependencies (List module dependencies in standard form)
/sourceDependencies (List all source-level dependencies)

`/std` (Specify Language Standard Version)

Article • 01/30/2025

Enable supported C and C++ language features from the specified version of the C or C++ language standard.

Syntax

> `/std:c++14`
> `/std:c++17`
> `/std:c++20`
> `/std:c++23preview`
> `/std:c++latest`
> `/std:c11`
> `/std:c17`
> `/std:clatest`

Remarks

The `/std` options are available in Visual Studio 2017 and later. They're used to control the version-specific ISO C or C++ programming language standard features enabled during compilation of your code. The options allow you to disable support for certain new language and library features: ones that may break your existing code that conforms to a particular version of the language standard.

The Microsoft C++ compiler in Visual Studio 2017 and later versions doesn't support C++ standards modes earlier than C++14 (`/std:c++14`). Such support isn't planned. As an imperfect workaround, it's possible to use older Visual C++ compiler toolsets that didn't implement features from later standards. For more information on how to install and use older compiler toolsets in Visual Studio, see Use native multi-targeting in Visual Studio to build old projects.

C++ standards support

Detect whether the `/std` option is in effect during a C++ compilation with the _MSVC_LANG preprocessor macro. For more information, see Preprocessor Macros.

> **Important**
>
> Because some existing code depends on the value of the macro `__cplusplus` being `199711L`, the MSVC compiler doesn't change the value of this macro unless you explicitly opt in by setting /Zc:__cplusplus. Specify `/Zc:__cplusplus` and the `/std` option to set `__cplusplus` to the appropriate value.

`/std:c++14`

Enables C++14 standard-specific features implemented by the MSVC compiler. This option is the default for code compiled as C++. It's available starting in Visual Studio 2015 Update 3.

This option disables compiler and standard library support for features that are changed or new in more recent versions of the language standard. However, it doesn't disable some C++17 features already implemented in previous releases of the MSVC compiler. For more information, see Microsoft C/C++ language conformance. The tables indicate which C++17 features are enabled when you specify `/std:c++14`.

The following features remain enabled when the `/std:c++14` option is specified to avoid breaking changes for users who took dependencies on features available in or before Visual Studio 2015 Update 2:

- Rules for auto with braced-init-lists
- typename in template template-parameters
- Removing trigraphs
- Attributes for namespaces and enumerators
- u8 character literals

`/std:c++17`

Enables C++17 standard-specific features and behavior. It enables the full set of C++17 features implemented by the MSVC compiler. This option disables compiler and standard library support for features that are new or changed after C++17. It specifically disables post-C++17 changes in the C++ Standard and versions of the Working Draft. It doesn't disable retroactive defect updates of the C++ Standard. This option is available starting in Visual Studio 2017 version 15.3.

Depending on the MSVC compiler version or update level, C++17 features may not be fully implemented or fully conforming when you specify the `/std:c++17` option. For an

overview of C++ language conformance in Visual C++ by release version, see Microsoft C/C++ language conformance.

`/std:c++20`

Enables C++20 standard-specific features and behavior.

Enables the standard conformance mode provided by `/permissive-` unless explicitly overridden with `/permissive`.

`/std:c++23preview`

Enables preview C++23 standard-specific features and behavior. Available starting in Visual Studio 2022 version 17.13 Preview 4. Preview features may change and may not be ABI compatible across releases.

This switch will be removed when the `/std:c++23` switch is implemented--at which point C++23 features will be fully implemented and ABI stable. If in project properties **C/C++** > **Language** you specify **Preview - ISO C++ 23 Standard (/std:c++preview)**, it will automatically change to mean `/std:c++23` once the new switch is implemented.

This switch differs from `/std:c++latest` in that it only enables features that are part of the C++23 standard. It doesn't enable experimental or in-progress features.

`/std:c++latest`

Enables all currently implemented compiler and standard library features proposed in the next ISO C++ working draft, as well as some in-progress and experimental features. This option is available starting with Visual Studio 2015 Update 3.

Depending on the MSVC compiler version or update level, features from published C++ standards or proposed features in the current C++ working draft, may not be fully implemented or fully conforming when you specify the `/std:c++latest` option. We recommend you use the latest version of Visual Studio for maximum standards conformance. For an overview of C++ language and library conformance in Visual C++ by release version, see Microsoft C/C++ language conformance.

Since Visual Studio 2019 version 16.8, the `/std:c++latest` option has enabled the standard conformance mode provided by `/permissive-` unless explicitly overridden with `/permissive`.

For a list of supported language and library features, see What's New for C++ in Visual Studio.

The `/std:c++latest` option doesn't enable features guarded by the `/experimental` switch, but may be required to enable them.

> ⓘ Note
>
> The compiler and library features enabled by `/std:c++latest` may appear in a future C++ standard. Features that haven't been approved are subject to breaking changes or removal without notice and are provided on an as-is basis.

C standards support

You can invoke the Microsoft C compiler by using the /TC or /Tc compiler option. It's used by default for code that has a `.c` file extension, unless overridden by a `/TP` or `/Tp` option. The default C compiler (that is, the compiler when `/std:c11` or `/std:c17` isn't specified) implements ANSI C89, but includes several Microsoft extensions, some of which are part of ISO C99. Some Microsoft extensions to C89 can be disabled by using the /Za compiler option, but others remain in effect. It isn't possible to specify strict C89 conformance. The compiler doesn't implement several required features of C99, so it isn't possible to specify C99 conformance, either.

`/std:c11`

Enables ISO C11 conformance. It's available starting in Visual Studio 2019 version 16.8.

`/std:c17`

Enables ISO C17 conformance. It's available starting in Visual Studio 2019 version 16.8.

Because the new preprocessor is needed to support these standards, the `/std:c11` and `/std:c17` compiler options set the /Zc:preprocessor option automatically. If you want to use the traditional (legacy) preprocessor for C11 or C17, you must set the `/Zc:preprocessor-` compiler option explicitly. Setting the `/Zc:preprocessor-` option may lead to unexpected behavior, and isn't recommended.

> ⓘ Note

At the time of release and through Visual Studio 2019 version 16.10, the Windows SDK and UCRT libraries installed by Visual Studio don't yet support C11 and C17 code. An updated version of the Windows SDK and UCRT is required. For more information and installation instructions, see **Install C11 and C17 support in Visual Studio**.

When you specify `/std:c11` or `/std:c17`, MSVC supports all the features of C11 and C17 required by the standards. The `/std:c11` and `/std:c17` compiler options enable support for these functionalities:

- _Pragma
- restrict
- _Noreturn and <stdnoreturn.h>
- _Alignas, _Alignof and <stdalign.h>
- _Generic and <tgmath.h>
- _Static_assert

The IDE uses C settings for IntelliSense and code highlighting when your source files have a `.c` file extension, or when you specify the /TC or /Tc compiler option. Currently, IntelliSense in C highlights keywords `_Alignas`, `_Alignof`, `_Noreturn`, and `_Static_assert`, but not the equivalent macros defined in the standard headers: `alignas`, `alignof`, `noreturn`, and `static_assert`.

Since C17 is largely a bug-fix release of ISO C11, MSVC support for C11 already includes all the relevant defect reports. There are no differences between the C11 and C17 versions except for the `__STDC_VERSION__` macro. It expands to `201112L` for C11, and `201710L` for C17.

The compiler doesn't support most optional features of ISO C11. Several of these optional features of C11 were required features of C99 that MSVC hasn't implemented for architectural reasons. You can use the feature test macros such as `__STDC_NO_VLA__` to detect compiler support levels for individual features. For more information about C-specific predefined macros, see Predefined macros.

- There's no conforming multithreading, atomic, or complex number support.
- `aligned_alloc` support is missing, because of the Windows heap implementation. The alternative is to use _aligned_malloc.
- Defect report 400 support is currently unimplemented for `realloc` because this change would break the ABI.

- Variable length array (VLA) support isn't planned. VLAs provide attack vectors comparable to gets, which is deprecated and planned for removal.

`/std:clatest`

The `/std:clatest` option behaves like the `/std:c++latest` switch for the C++ compiler. The switch enables all currently implemented compiler and standard library features proposed in the next draft C standard, as well as some in-progress and experimental features.

For more information, see the C Standard library features section of Microsoft C/C++ language conformance.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Language** property page.
3. In **C++ Language Standard** (or for C, **C Language Standard**), choose the language standard to support from the dropdown control, then choose **OK** or **Apply** to save your changes.

See also

/Zc:__cplusplus[-]
MSVC compiler options
MSVC compiler command-line syntax

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

/Tc, /Tp, /TC, /TP (Specify Source File Type)

Article • 08/03/2021

The **/Tc** option specifies that its filename argument is a C source file, even if it does not have a .c extension. The **/Tp** option specifies that its filename argument is a C++ source file, even if it doesn't have a .cpp or .cxx extension. A space between the option and the filename is optional. Each option specifies one file; to specify additional files, repeat the option.

/TC and **/TP** are global variants of **/Tc** and **/Tp**. They specify to the compiler to treat all files named on the command line as C source files (**/TC**) or C++ source files (**/TP**), without regard to location on the command line in relation to the option. These global options can be overridden on a single file by means of **/Tc** or **/Tp**.

Syntax

> **/Tc** *filename*
> **/Tp** *filename*
> **/TC**
> **/TP**

Arguments

filename
A C or C++ source file.

Remarks

By default, **CL** assumes that files with the .c extension are C source files and files with the .cpp or the .cxx extension are C++ source files.

When either the **TC** or **Tc** option is specified, any specification of the /Zc:wchar_t (wchar_t Is Native Type) option is ignored.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Advanced** property page.
3. Modify the **Compile As** property. Choose **OK** or **Apply** to apply your changes.

To set this compiler option programmatically

- See CompileAs.

Examples

This CL command line specifies that MAIN.c, TEST.prg, and COLLATE.prg are all C source files. CL will not recognize PRINT.prg.

```
CL MAIN.C /TcTEST.PRG /TcCOLLATE.PRG PRINT.PRG
```

This CL command line specifies that TEST1.c, TEST2.cxx, TEST3.huh, and TEST4.o are compiled as C++ files, and TEST5.z is compiled as a C file.

```
CL TEST1.C TEST2.CXX TEST3.HUH TEST4.O /Tc TEST5.Z /TP
```

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/translateInclude`

Article • 02/18/2022

This switch instructs the compiler to treat `#include` as `import` for header files that have been built into header unit (`.ifc`) files and that are specified on the command line with /headerUnit.

When used with /scanDependencies or /sourceDependencies-directives, the compiler lists as imported header units in the generated dependency file those headers that are both included in the source and have a corresponding entry in a `header-units.json` file. This dependency info is used by the build system to generate compiled header unit `.ifc` files. Once the header units are built, they're treated by the compiler as an `import` instead of an `#include`.

The `header-units.json` file is only consulted when `/translateInclude` is specified. For more information about the format and purpose of the `header-units.json` file, see header-units.json.

If an `#include` file isn't listed in the `header-units.json` file, it's treated as a normal `#include`.

For an example of how this switch is used, see Walkthrough: Build and import header units in Microsoft Visual C++.

Syntax

> **`/translateInclude`**

Remarks

`/translateInclude` is available in Visual Studio 2019 version 16.10, or later.

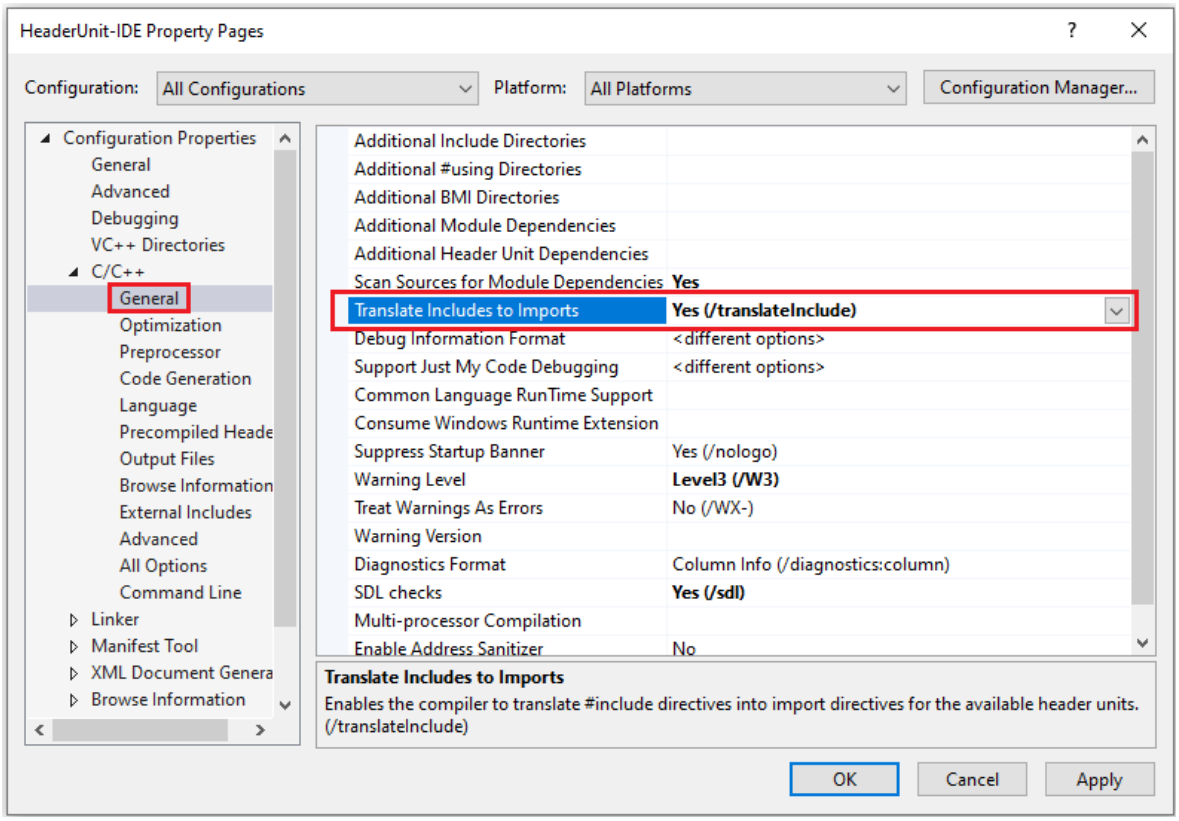
`/translateInclude` requires /std:c++20 or later.

To set this compiler option in Visual Studio

To enable `/translateInclude`, in the project properties dialog, set **Translate Includes to Imports**:

1. In the left-hand pane of the project property pages, select **Configuration Properties** > **C/C++** > **General**.

2. Change the **Translate Includes to Imports** dropdown to **Yes**.



3. Choose **OK** or **Apply** to save your changes.

See also

/headerUnit (Use header unit IFC).
/exportHeader (Create header units)
/reference (Use named module IFC)
/scanDependencies
/sourceDependencies-directives
Walkthrough: Build and import header units in Microsoft Visual C++

/U, /u (Undefine symbols)

Article • 08/03/2021

The `/U` compiler option undefines the specified preprocessor symbol. The `/u` compiler option undefines the Microsoft-specific symbols that the compiler defines.

Syntax

> `/U`[]*symbol*
> `/u`

Arguments

symbol
The preprocessor symbol to undefine.

Remarks

Neither of the `/U` and `/u` options can undefine a symbol created by using the `#define` directive.

The `/U` option can undefine a symbol that was previously defined by using the `/D` option.

By default, the compiler may define a large number of Microsoft-specific symbols. Here are a few common ones:

Symbol	Function
`_CHAR_UNSIGNED`	Default char type is unsigned. Defined when the /J option is specified.
`_CPPRTTI`	Defined for code compiled with the /GR option.
`_CPPUNWIND`	Defined for code compiled with the /EHsc option.
`_DLL`	Defined when the /MD option is specified.
`_M_IX86`	By default, defined to 600 for x86 targets.
`_MSC_VER`	Defined as a unique integer value for each compiler version. For more information, see Predefined macros.

Symbol	Function
`_WIN32`	Defined for WIN32 applications. Always defined.
`_MT`	Defined when the /MD or /MT option is specified.

For a complete list of Microsoft-specific predefined macros, see Predefined macros.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Advanced** property page.
3. Modify the **Undefine Preprocessor Definitions** or **Undefine All Preprocessor Definitions** properties.

To set this compiler option programmatically

- See UndefineAllPreprocessorDefinitions or UndefinePreprocessorDefinitions.

See also

MSVC compiler options
MSVC compiler command-line syntax
`/J` (Default char type is unsigned)
`/GR` (Enable run-time type information)
`/EH` (Exception handling model)
`/MD`, `/MT`, `/LD` (Use run-time library)

`/utf-8` (Set source and execution character sets to UTF-8)

Article • 02/01/2022

Specifies both the source character set and the execution character set as UTF-8.

Syntax

> `/utf-8`

Remarks

You can use the `/utf-8` option to specify both the source and execution character sets as encoded by using UTF-8. It's equivalent to specifying `/source-charset:utf-8` `/execution-charset:utf-8` on the command line. Any of these options also enables the `/validate-charset` option by default. For a list of supported code page identifiers and character set names, see Code Page Identifiers.

By default, Visual Studio detects a byte-order mark to determine if the source file is in an encoded Unicode format, for example, UTF-16 or UTF-8. If no byte-order mark is found, it assumes that the source file is encoded in the current user code page, unless you've specified a code page by using `/utf-8` or the `/source-charset` option. Visual Studio allows you to save your C++ source code in any of several character encodings. For information about source and execution character sets, see Character sets in the language documentation.

Set the option in Visual Studio or programmatically

To set this compiler option in the Visual Studio development environment

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

3. In **Additional Options**, add the `/utf-8` option to specify your preferred encoding.

4. Choose **OK** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax
/execution-charset (Set execution character set)
/source-charset (Set source character set)
/validate-charset (Validate for compatible characters)

/V (Version Number)

Article • 08/03/2021

Deprecated. Embeds a text string in the .obj file.

Syntax

```
/Vstring
```

Arguments

string
A string specifying the version number or copyright notice to be embedded in an .obj file.

Remarks

The stringcan label an .obj file with a version number or a copyright notice. Any space or tab characters must be enclosed in double quotation marks (") if they are a part of the string. A backslash (\) must precede any double quotation marks if they are a part of the string. A space between **/V** and `string` is optional.

You can also use comment (C/C++) with the compiler comment-type argument to place the name and version number of the compiler in the .obj file.

The **/V** option is deprecated beginning in Visual Studio 2005; **/V** was primarily used to support building virtual device drivers (VxDs), and building VxDs is no longer supported by the Visual C++ toolset. For a list of deprecated compiler options, see **Deprecated and Removed Compiler Options** in Compiler Options Listed by Category.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/validate-charset` (Validate for compatible characters)

Article • 02/01/2022

This compiler option validates that the source file text contains only characters representable as UTF-8.

Syntax

> `validate-charset`[`-`]

Remarks

You can use the `/validate-charset` option to validate that the source code contains only characters that can be represented in both the source character set and the execution character set. This check is enabled automatically when you specify `/source-charset`, `/execution-charset`, or `/utf-8` compiler options. To explicitly disable this check, specify the `/validate-charset-` option.

By default, Visual Studio detects a byte-order mark to determine if the source file is in an encoded Unicode format, for example, UTF-16 or UTF-8. If no byte-order mark is found, it assumes that the source file is encoded in the current user code page, unless you have specified a code page by using `/utf-8` or the `/source-charset` option. Visual Studio allows you to save your C++ source code in any of several character encodings. For information about source and execution character sets, see Character sets in the language documentation. For a list of supported code page identifiers and character set names, see Code Page Identifiers.

Visual Studio uses UTF-8 as the internal character encoding during conversion between the source character set and the execution character set. If a character in the source file can't be represented in the execution character set, the UTF-8 conversion substitutes a question mark (`?`) character. If a substitution occurs, the `/validate-charset` option causes the compiler to report a warning.

To set this compiler option in the Visual Studio development environment

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In **Additional Options**, add the `/validate-charset` or `/validate-charset-` option.
4. Choose **OK** to save your changes.

See also

MSVC compiler options
MSVC compiler command-line syntax
/execution-charset (Set execution character set)
/source-charset (Set source character set)
/utf-8 (Set source and execution character sets to UTF-8)

/vd (Disable Construction Displacements)

Article • 08/03/2021

Syntax

```
/vdn
```

Arguments

0
Suppresses the vtordisp constructor/destructor displacement member. Choose this option only if you are certain that all class constructors and destructors call virtual functions virtually.

1
Enables the creation of hidden vtordisp constructor/destructor displacement members. This choice is the default.

2
Allows you to use dynamic_cast Operator on an object being constructed. For example, a dynamic_cast from a virtual base class to a derived class.

/vd2 adds a vtordisp field when you have a virtual base with virtual functions. **/vd1** should be sufficient. The most common case where **/vd2** is necessary is when the only virtual function in your virtual base is a destructor.

Remarks

These options apply only to C++ code that uses virtual bases.

Visual C++ implements C++ construction displacement support in situations where virtual inheritance is used. Construction displacements solve the problem created when a virtual function, declared in a virtual base and overridden in a derived class, is called from a constructor during construction of a further derived class.

The problem is that the virtual function may be passed an incorrect `this` pointer as a result of discrepancies between the displacements to the virtual bases of a class and the displacements to its derived classes. The solution provides a single construction displacement adjustment, called a vtordisp field, for each virtual base of a class.

By default, vtordisp fields are introduced whenever the code defines user-defined constructors and destructors and also overrides virtual functions of virtual bases.

These options affect entire source files. Use vtordisp to suppress and then re-enable vtordisp fields on a class-by-class basis.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/vlen`

Article • 10/16/2024

Specifies the vector length for code generation on x86 and x64. For more information about `/arch` for x86 and x64, see /arch (x86) and /arch (x64).

Syntax

> `/vlen=`[`256`|`512`]

> `/vlen`

Arguments

`/vlen=256`
Specify a vector length of 256 bits for auto-vectorization and other optimizations.

`/vlen=512`
Specify a vector length of 512 bits for auto-vectorization and other optimizations.

`/vlen`
Specify the default vector length for the selected `/arch` setting.

Remarks

If a specific `/vlen` value isn't specified, the default vector length depends on the `/arch` flag setting. The `/vlen` flag can override the default vector length specified by `/arch:AVX512` or `/arch:AVX10.1` flag. For example:

- `/arch:AVX512 /vlen=256` overrides the default vector length of 512 bits specified by `/arch:AVX512` to be 256 bits.
- `/arch:AVX10.1 /vlen=512` overrides the default vector length of 256 bits specified by `/arch:AVX10.1` to be 512 bits.

When the specified `/vlen` value is incompatible with specified `/arch` flag, a warning is generated and default vector length for the `/arch` setting is used. For example:

- `/arch:AVX2 /vlen=512` generates a warning because AVX2 doesn't support 512-bit vectors. Vector length of 256 bits is used in this case.

To set the `/vlen=256` or `/vlen=512` compiler option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In the **Additional options** box, add `/vlen=256` or `/vlen=512`. Choose **OK** to save your changes.

See also

/arch (Minimum CPU Architecture)
MSVC compiler options
MSVC compiler command-line syntax

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/vmb`, `/vmg` (Representation method)

Article • 08/03/2021

Select the method that the compiler uses to represent pointers to class members.

Syntax

> `/vmb`
>
> `/vmg`

Options

`/vmb` is the compiler's default behavior. Its behavior is the same as `#pragma pointers_to_members(best_case)`. It doesn't require or ensure complete types. For complete types, it uses the best representation among single, multiple, or virtual inheritance based the inheritance of the class type. For incomplete types, it uses the largest, most general representation.

`/vmg` lets you specify compiler behavior in combination with /vmm, /vms, /vmv (General purpose representation) to declare a pointer to a member of a class before defining the class. This need can arise if you define members in two different classes that reference each other. For such mutually referencing classes, one class must be referenced before it's defined.

Remarks

You can also use #pragma pointers_to_members or Inheritance keywords in your code to specify a pointer representation.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/vmm`, `/vms`, `/vmv` (General Purpose Representation)

Article • 08/06/2021

Used when /vmg is selected as the representation method. These options indicate the inheritance model of the not-yet-encountered class definition.

Syntax

> `/vmm`
>
> `/vms`
>
> `/vmv`

Options

`/vmm`
Specifies the most general representation of a pointer to a member of a class as one that uses multiple inheritance.

The corresponding inheritance keyword and argument to #pragma pointers_to_members is `multiple_inheritance`.

This representation is larger than the one required for single inheritance.

If you use `/vmm` and declare a pointer to member of a class that has a virtual inheritance model, the compiler generates an error.

`/vms`
Specifies the most general representation of a pointer to a member of a class as one that uses either no inheritance or single inheritance.The corresponding inheritance keyword and argument to #pragma pointers_to_members is `single_inheritance`.

This option generates the smallest possible representation of a pointer to a member of a class.

If you use `/vms` and declare a pointer to member of a class that has a multiple or virtual inheritance model, the compiler generates an error.

`/vmv`
Specifies the most general representation of a pointer to a member of a class as one

that uses virtual inheritance. This pointer representation never causes an error and is the default.

The corresponding inheritance keyword and argument to #pragma pointers_to_members is `virtual_inheritance`.

This option requires a larger pointer and more code to interpret the pointer than the other options.

Remarks

In Visual Studio 2019 and earlier versions, Microsoft uses different representations (of different sizes) for pointer-to-member types. Pointers to members of classes that have no inheritance or single inheritance have the smallest representation. Pointers to members of classes that have multiple inheritance are larger. Pointers to members of classes that have virtual inheritance are the largest. When no representation model is specified to the compiler, it defaults to using the largest, most general representation.

When you specify one of these inheritance-model options, that model gets used for all pointers to class members, no matter their inheritance type or whether you declare the pointer before or after the class. If you always use single-inheritance classes, you can reduce code size by compiling with `/vms`. However, if you want to use the most general case (at the expense of the largest data representation), compile with `/vmv`.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/vmb, /vmg (Representation method)
MSVC compiler options
MSVC compiler command-line syntax

/volatile (volatile Keyword Interpretation)

Article • 06/15/2022

Specifies how the volatile keyword is to be interpreted.

Syntax

> **/volatile:{iso|ms}**

Arguments

/volatile:iso
Selects strict `volatile` semantics as defined by the ISO-standard C++ language. Acquire/release semantics are not guaranteed on volatile accesses. If the compiler targets ARM (except ARM64EC), this is the default interpretation of `volatile`.

/volatile:ms
Selects Microsoft extended `volatile` semantics, which add memory ordering guarantees beyond the ISO-standard C++ language. Acquire/release semantics are guaranteed on volatile accesses. However, this option also forces the compiler to generate hardware memory barriers, which might add significant overhead on ARM and other weak memory-ordering architectures. If the compiler targets ARM64EC or any non-ARM platform, this is default interpretation of `volatile`.

Remarks

We strongly recommend that you use **/volatile:iso** along with explicit synchronization primitives and compiler intrinsics when you are dealing with memory that is shared across threads. For more information, see volatile.

If you port existing code or change this option in the middle of a project, it may be helpful to enable warning C4746 to identify code locations that are affected by the difference in semantics.

There is no `#pragma` equivalent to control this option.

To set the /volatile compiler option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In the **Additional options** box, add **/volatile:iso** or **/volatile:ms** and then choose **OK** or **Apply** to save your changes.

See also

volatile
MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/volatileMetadata` (Generate metadata on volatile memory accesses)

Article • 05/31/2024

Generate metadata for volatile memory accesses to improve performance when running x64 code on ARM64.

Syntax

```
C++

/volatileMetadata[-]
```

Arguments

`-`

Turns off `/volatileMetadata`. This may result in worse performance when your code runs in emulation mode on ARM64 because the emulator pessimistically assumes that every load/store needs a barrier.

Remarks

Starting with Visual Studio 2019 16.10, `/volatileMetadata` is on by default when generating x64 code. It improves the emulation performance of x64 code on ARM64 by generating metadata that identifies volatile memory addresses. An emulator can use this metadata to improve performance by not using acquire/release semantics on those accesses it knows aren't volatile. Without this metadata, the emulator assumes that all addresses are volatile and uses acquire and release semantics.

One side effect of `/volatileMetadata` is you may see `npad` macros used in the generated code. This macro expands to a specified number of `NOP` instructions that create an address to associate with a memory barrier. That address is then recorded in the metadata to indicate that acquire/release semantics should be used to access it.

`/volatileMetadata` is ignored when targeting x86.

`/volatileMetadata` can be disabled by using `/volatileMetadata-`.

Requirements

Visual Studio 2019 version 16.10 or later.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

/w, /W0, /W1, /W2, /W3, /W4, /w1, /w2, /w3, /w4, /Wall, /wd, /we, /wo, /Wv, /WX (Warning level)

Article • 02/18/2022

Specifies how the compiler generates warnings for a given compilation.

Syntax

> **/w**
> **/W0**
> **/W1**
> **/W2**
> **/W3**
> **/W4**
> **/Wall**
> **/Wv**[**:***version*]
> **/WX**
> **/w1***warning*
> **/w2***warning*
> **/w3***warning*
> **/w4***warning*
> **/wd***warning*
> **/we***warning*
> **/wo***warning*

Remarks

The warning options specify which compiler warnings to display and the warning behavior for the entire compilation.

The warning options and related arguments are described in the following tables:

Option	Description
/w	Suppresses all compiler warnings.

Option	Description
/W0 **/W1** **/W2** **/W3** **/W4**	Specifies the level of warnings to be generated by the compiler. Valid warning levels range from 0 to 4: **/W0** suppresses all warnings. It's equivalent to **/w**. **/W1** displays level 1 (severe) warnings. **/W1** is the default setting in the command-line compiler. **/W2** displays level 1 and level 2 (significant) warnings. **/W3** displays level 1, level 2, and level 3 (production quality) warnings. **/W3** is the default setting in the IDE. **/W4** displays level 1, level 2, and level 3 warnings, and all level 4 (informational) warnings that aren't off by default. We recommend that you use this option to provide lint-like warnings. For a new project, it may be best to use **/W4** in all compilations. This option helps ensure the fewest possible hard-to-find code defects.
/Wall	Displays all warnings displayed by **/W4** and all other warnings that **/W4** doesn't include—for example, warnings that are off by default. For more information, see Compiler warnings that are off by default.
/Wv[:*version*]	Displays only warnings introduced in the *version* compiler version and earlier. You can use this option to suppress new warnings in code when you migrate to a newer version of the compiler. It lets you maintain your existing build process while you fix them. The optional parameter *version* takes the form *nn*[*.mm*[*.bbbbb*]], where *nn* is the major version number, *mm* is the optional minor version number, and *bbbbb* is the optional build number of the compiler. For example, use **/Wv:17** to display only warnings introduced in Visual Studio 2012 (major version 17) or earlier. That is, it displays warnings from any version of the compiler that has a major version number of 17 or less. It suppresses warnings introduced in Visual Studio 2013 (major version 18) and later. By default, **/Wv** uses the current compiler version number, and no warnings are suppressed. For information about which warnings are suppressed by compiler version, see Compiler warnings by compiler version.
/WX	Treats all compiler warnings as errors. For a new project, it may be best to use **/WX** in all compilations; resolving all warnings ensures the fewest possible hard-to-find code defects. The linker also has a **/WX** option. For more information, see /WX (Treat linker warnings as errors).

The following options are mutually exclusive with each other. The last option that's specified from this group is the one applied:

Option	Description

Option	Description
/w1*nnnn* **/w2***nnnn* **/w3***nnnn* **/w4***nnnn*	Sets the warning level for the warning number specified by *nnnn*. These options let you change the compiler behavior for that warning when a specific warning level is set. You can use these options in combination with other warning options to enforce your own coding standards for warnings, rather than the default ones provided by Visual Studio. For example, **/w34326** causes C4326 to be generated as a level 3 warning instead of level 1. If you compile by using both the **/w34326** option and the **/W2** option, warning C4326 isn't generated.
/wd*nnnn*	Suppresses the compiler warning that is specified by *nnnn*. For example, **/wd4326** suppresses compiler warning C4326.
/we*nnnn*	Treats the compiler warning that is specified by *nnnn* as an error. For example, **/we4326** causes warning number C4326 to be treated as an error by the compiler.
/wo*nnnn*	Reports the compiler warning that is specified by *nnnn* only once. For example, **/wo4326** causes warning C4326 to be reported only once, the first time it's encountered by the compiler.

If you use any warning options when you create a precompiled header, it keeps those settings. Using the precompiled header puts those same warning options in effect again. To override the precompiled header warning options, set another warning option on the command line.

You can use a #pragma warning directive to control the level of warning that's reported at compile time in specific source files.

Warning pragma directives in source code are unaffected by the **/w** option.

The build errors documentation describes the warnings and warning levels, and indicates why certain statements may not compile as you intend.

To set the compiler options in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. To set the **/W0**, **/W1**, **/W2**, **/W3**, **/W4**, **/Wall**, **/Wv**, **/WX**, or **/WX-** options, select **Configuration Properties** > **C/C++** > **General**.

 - To set the **/W0**, **/W1**, **/W2**, **/W3**, **/W4**, or **/Wall** options, modify the **Warning Level** property.
 - To set the **/WX** or **/WX-** options, modify the **Treat Warnings as Errors** property.
 - To set the version for the **/Wv** option, enter the compiler version number in the **Warning Version** property.

3. To set the **/wd** or **/we** options, select the **Configuration Properties** > **C/C++** > **Advanced** property page.

 - To set the **/wd** option, select the **Disable Specific Warnings** property dropdown control and then choose **Edit**. In the edit box in the **Disable Specific Warnings** dialog, enter the warning number. To enter more than one warning, separate the values by using a semicolon (;). For example, to disable both C4001 and C4010, enter **4001;4010**. Choose **OK** to save your changes and return to the **Property Pages** dialog.
 - To set the **/we** option, Select the **Treat Specific Warnings As Errors** property dropdown control and then choose **Edit**. In the edit box in the **Treat Specific Warnings As Errors** dialog, enter the warning number. To enter more than one warning, separate the values by using a semicolon (;). For example, to treat both C4001 and C4010 as errors, enter **4001;4010**. Choose **OK** to save your changes and return to the **Property Pages** dialog.

4. To set the **/wo** option, select the **Configuration Properties** > **C/C++** > **Command Line** property page. Enter the compiler option in the **Additional Options** box.

5. Choose **OK** to save your changes.

To set the compiler option programmatically

- See WarningLevel, WarnAsError, DisableSpecificWarnings, and AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax

/WL (Enable One-Line Diagnostics)

Article • 08/03/2021

Appends additional information to an error or warning message.

Syntax

```
/WL
```

Remarks

Error and warning messages from the C++ compiler can be followed by additional information that appears, by default, on a new line. When you compile from the command line, the extra line of information can be appended to the error or warning message. It's useful if you capture your build output to a log file and then process that log to find all errors and warnings. A semicolon will separate the error or warning message from the additional line.

Not all error and warning messages have an extra line of information. The following code will generate an error that has another line of information. It lets you test the effect when you use **/WL**.

C++

```
// compiler_option_WL.cpp
// compile with: /WL
#include <queue>
int main() {
   std::queue<int> q;
   q.fromthecontinuum();   // C2039
}
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.

3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Wp64 (Detect 64-Bit Portability Issues)

Article • 08/03/2021

This compiler option is obsolete. In versions of Visual Studio before Visual Studio 2013, this detects 64-bit portability problems on types that are also marked with the __w64 keyword.

Syntax

```
/Wp64
```

Remarks

By default, in versions of Visual Studio before Visual Studio 2013, the **/Wp64** compiler option is off in the MSVC compiler that builds 32-bit x86 code, and on in the MSVC compiler that builds 64-bit, x64 code.

> **ⓘ Important**
>
> The **/Wp64** compiler option and **__w64** keyword are deprecated in Visual Studio 2010 and Visual Studio 2012, and not supported starting in Visual Studio 2013. If you convert a project that uses this switch, the switch will not be migrated during conversion. To use this option in Visual Studio 2010 or Visual Studio 2012, you must type the compiler switch under **Additional Options** in the **Command Line** section of the project properties. If you use the **/Wp64** compiler option on the command line, the compiler issues Command-Line Warning D9002. Instead of using this option and keyword to detect 64-bit portability issues, use a MSVC compiler that targets a 64-bit platform and specify the **/W4** option. For more information, see **Configure C++ projects for 64-bit, x64 targets**.

Variables of the following types are tested on a 32-bit operating system as if they were being used on a 64-bit operating system:

- int
- long

- pointer

If you regularly compile your application by using a compiler that builds 64-bit, x64 code, you can just disable **/Wp64** in your 32-bit compilations because the 64-bit compiler will detect all issues. For more information about how to target a Windows 64-bit operating system, see Configure C++ projects for 64-bit, x64 targets.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** box to include **/Wp64**.

To set this compiler option programmatically

- See Detect64BitPortabilityProblems.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Configure C++ projects for 64-bit, x64 targets

`/X` (Ignore standard include paths)

Article • 08/03/2021

Prevents the compiler from searching for include files in directories specified in the `PATH` and `INCLUDE` environment variables.

Syntax

> `/X`

Remarks

You can use this option with the /I (Additional include directories) option to specify an alternate path to the standard include files.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Preprocessor** property page.
3. Modify the **Ignore Standard Include Path** property.

To set this compiler option programmatically

- See IgnoreStandardIncludePath.

Example

In the following command, `/X` tells the compiler to ignore locations specified by the `PATH` and `INCLUDE` environment variables, and `/I` specifies the directory to look in for include files:

Windows Command Prompt

```
CL /X /I \ALT\INCLUDE MAIN.C
```

See also

MSVC compiler options
MSVC compiler command-line syntax

/Y (precompiled headers)

Article • 08/03/2021

The following compiler options affect the generation and use of precompiled headers:

- /Y- (Ignore precompiled header options)
- /Yc (Create precompiled header file)
- /Yd (Place Debug Information in Object file)
- /Yl (Inject PCH reference for debug library)
- /Yu (Use precompiled header file)

For details on working with precompiled headers, see precompiled header files.

See also

MSVC compiler options
MSVC compiler command-line syntax

/Y- (Ignore Precompiled Header Options)

Article • 08/03/2021

Causes all other `/Y` compiler options to be ignored (and cannot itself be overridden).

Syntax

```
/Y-
```

Remarks

For more information on precompiled headers, see:

- /Y (Precompiled Headers)
- Precompiled Header Files

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Yc (Create Precompiled Header File)

Article • 08/03/2021

Instructs the compiler to create a precompiled header (.pch) file that represents the state of compilation at a certain point.

Syntax

> **/Yc**
> **/Yc***filename*

Arguments

filename
Specifies a header (.h) file. When this argument is used, the compiler compiles all code up to and including the .h file.

Remarks

When **/Yc** is specified without an argument, the compiler compiles all code up to the end of the base source file, or to the point in the base file where a hdrstop directive occurs. The resulting .pch file has the same base name as your base source file unless you specify a different file name using the **hdrstop** pragma or the **/Fp** option.

The precompiled code is saved in a file with a name created from the base name of the file specified with the **/Yc** option and a .pch extension. You can also use the /Fp (Name .Pch File) option to specify a name for the precompiled header file.

If you use **/Yc***filename*, the compiler compiles all code up to and including the specified file for subsequent use with the /Yu (Use Precompiled Header File) option.

If the options **/Yc***filename* and **/Yu***filename* occur on the same command line and both reference, or imply, the same file name, **/Yc***filename* takes precedence. This feature simplifies the writing of makefiles.

For more information on precompiled headers, see:

- /Y (Precompiled Headers)
- Precompiled Header Files

To set this compiler option in the Visual Studio development environment

1. Select a .cpp file. The .cpp file must #include the .h file that contains precompiled header information. The project's **/Yc** setting can be overridden at the file level.
2. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
3. Open the **Configuration Properties**, **C/C++**, **Precompiled Headers** property page.
4. Modify the **Precompiled Header** property.
5. To set the filename, modify the **Precompiled Header File** property.

To set this compiler option programmatically

- See PrecompiledHeaderThrough and UsePrecompiledHeader.

Example

Consider the following code:

```
C++

// prog.cpp
// compile with: cl /c /Ycmyapp.h prog.cpp
#include <afxwin.h>   // Include header for class library
#include "resource.h" // Include resource definitions
#include "myapp.h"    // Include information specific to this app
// ...
```

When this code is compiled with the command `CL /YcMYAPP.H PROG.CPP`, the compiler saves all the preprocessing for AFXWIN.h, RESOURCE.h, and MYAPP.h in a precompiled header file called MYAPP.pch.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
Precompiled Header Files

/Yd (Place Debug Information in Object File)

Article • 08/03/2021

Paces complete debugging information in all object files created from a precompiled header (.pch) file when used with the /Yc and /Z7 options. Deprecated.

Syntax

```
/Yd
```

Remarks

/Yd is deprecated; Visual C++ now supports multiple objects writing to a single .pdb file, use **/Zi** instead. For a list of deprecated compiler options, see **Deprecated and Removed Compiler Options** in Compiler Options Listed by Category.

Unless you need to distribute a library containing debugging information, use the /Zi option rather than **/Z7** and **/Yd**.

Storing complete debugging information in every .obj file is necessary only to distribute libraries that contain debugging information. It slows compilation and requires considerable disk space. When **/Yc** and **/Z7** are used without **/Yd**, the compiler stores common debugging information in the first .obj file created from the .pch file. The compiler does not insert this information into .obj files subsequently created from the .pch file; it inserts cross-references to the information. No matter how many .obj files use the .pch file, only one .obj file contains the common debugging information.

Although this default behavior results in faster build times and reduces disk-space demands, it is undesirable if a small change requires rebuilding the .obj file containing the common debugging information. In this case, the compiler must rebuild all .obj files containing cross-references to the original .obj file. Also, if a common .pch file is used by different projects, reliance on cross-references to a single .obj file is difficult.

For more information on precompiled headers, see:

- /Y (Precompiled Headers)

- Precompiled Header Files

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

Examples

Suppose you have two base files, F.cpp and G.cpp, each containing these **#include** statements:

```
#include "windows.h"
#include "etc.h"
```

The following command creates the precompiled header file ETC.pch and the object file F.obj:

```
CL /YcETC.H /Z7 F.CPP
```

The object file F.obj includes type and symbol information for WINDOWS.h and ETC.h (and any other header files they include). Now you can use the precompiled header ETC.pch to compile the source file G.cpp:

```
CL /YuETC.H /Z7 G.CPP
```

The object file G.obj does not include the debugging information for the precompiled header but simply references that information in the F.obj file. Note that you must link with the F.obj file.

If your precompiled header was not compiled with **/Z7**, you can still use it in later compilations using **/Z7**. However, the debugging information is placed in the current object file, and local symbols for functions and types defined in the precompiled header are not available to the debugger.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Yl (Inject PCH Reference for Debug Library)

Article • 08/03/2021

The **/Yl** option generates a unique symbol in a precompiled header file, and a reference to this symbol is injected in all object files that use the precompiled header.

Syntax

> **/Yl**
> **/Yl***name*
> **/Yl-**

Arguments

name
An optional name used as part of the unique symbol.

-
A dash (-) explicitly disables the **/Yl** compiler option.

Remarks

The **/Yl** compiler option creates a unique symbol definition in a precompiled header file created by using the /Yc option. References to this symbol are automatically injected in all files that include the precompiled header by using the /Yu compiler option. The **/Yl** option is enabled by default when **/Yc** is used to create a precompiled header file.

The **/Yl***name* option is used to create an identifiable symbol in the precompiled header file. The compiler uses the *name* argument as part of the decorated symbol name it creates, similar to `__@@_PchSym_@00@...@name`, where the ellipsis (...) represents a unique compiler-generated character string. If the *name* argument is omitted, the compiler generates a symbol name automatically. Normally, you do not need to know the name of the symbol. However, when your project uses more than one precompiled header file, the **/Yl***name* option may be useful to determine which object files use which precompiled header. You can use *name* as a search string to find the symbol reference in a dump file.

/Yl- disables the default behavior and does not put an identifying symbol in the precompiled header file. Compiled files that include this precompiled header do not get a common symbol reference.

When **/Yc** is not specified, any **/Yl** option has no effect, but if specified it must match any **/Yl** option passed when **/Yc** is specified.

If you use **/Yl-**, **/Yc** and /Z7 options to build a precompiled header file, the debugging information is stored in the object file for the source file used to create the precompiled header, rather than a separate .pdb file. If this object file is then made part of a library, LNK1211 errors or LNK4206 warnings can occur in builds that use this library and the precompiled header file, if the source file used to create the precompiled header file does not define any symbols itself. The linker may exclude the object file from the link, along with the associated debugging information, when nothing in the object file is referenced in the library client. To solve this problem, specify **/Yl** (or remove the **/Yl-** option) when you use **/Yc** to create the precompiled header file. This ensures that the object file from the library that contains the debugging information gets linked in your build.

For more information on precompiled headers, see:

- /Y (Precompiled Headers)
- Precompiled Header Files

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Add the **/Yl***name* compiler option in the **Additional Options** box. Choose **OK** to save your changes.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options

MSVC Compiler Command-Line Syntax

`/Yu` (Use precompiled header file)

Article • 08/03/2021

Instructs the compiler to use an existing precompiled header (`.pch`) file in the current compilation.

Syntax

> `/Yu`[*filename*]

Arguments

filename
The name of a header file, which is included in the source file using a `#include` preprocessor directive.

Remarks

The name of the include file must be the same for both the `/Yc` option that creates the precompiled header and for any later `/Yu` option that indicates use of the precompiled header.

For `/Yc`, *filename* specifies the point at which precompilation stops; the compiler precompiles all code though *filename* and names the resulting precompiled header using the base name of the include file and an extension of `.pch`.

The `.pch` file must have been created using `/Yc`.

The compiler treats all code occurring before the .h file as precompiled. It skips to just beyond the `#include` directive associated with the `.h` file, uses the code contained in the `.pch` file, and then compiles all code after *filename*.

On the command line, no space is allowed between `/Yu` and *filename*.

When you specify the `/Yu` option without a file name, your source program must contain a #pragma hdrstop pragma that specifies the file name of the precompiled header, `.pch` file. In this case, the compiler will use the precompiled header (`.pch` file) named by /Fp (Name .pch file). The compiler skips to the location of that pragma and restores the compiled state from the specified precompiled header file. Then it compiles

only the code that follows the pragma. If `#pragma hdrstop` doesn't specify a file name, the compiler looks for a file with a name derived from the base name of the source file with a *`.pch`* extension. You can also use the **`/Fp`** option to specify a different *`.pch`* file.

If you specify the **`/Yu`** option without a file name and fail to specify a `hdrstop` pragma, an error message is generated and the compilation is unsuccessful.

If the **`/Yc`***filename* and **`/Yu`***filename* options occur on the same command line and both reference the same file name, **`/Yc`***filename* takes precedence, precompiling all code up to and including the named file. This feature simplifies the writing of makefiles.

Because *`.pch`* files contain information about the machine environment and memory address information about the program, you should only use a *`.pch`* file on the machine where it was created.

For more information on precompiled headers, see:

- /Y (Precompiled headers)
- Precompiled header files

To set this compiler option in the Visual Studio development environment

1. Specify /Yc (Create precompiled header file) on a .cpp file in your project.
2. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
3. Select the **Configuration Properties** > **C/C++** > **Precompiled Headers** property page.
4. Modify the **Precompiled Header** property, the **Create/Use PCH Through File** property, or the **Create/Use Precompiled Header** property.

To set this compiler option programmatically

- See PrecompiledHeaderThrough and UsePrecompiledHeader.

Example

If the following code:

C++

```
#include <afxwin.h>   // Include header for class library
#include "resource.h" // Include resource definitions
#include "myapp.h"    // Include information specific to this app
...
```

is compiled by using the command line `CL /YuMYAPP.H PROG.CPP`, the compiler doesn't process the three include statements. Instead, it uses precompiled code from *`MYAPP.pch`*, which saves the time involved in preprocessing all three of the files (and any files they might include).

You can use the /Fp (Name .pch file) option with the **`/Yu`** option to specify the name of the *`.pch`* file if the name is different from either the *filename* argument to **`/Yc`** or the base name of the source file, as in the following example:

Windows Command Prompt

```
CL /YuMYAPP.H /FpMYPCH.pch PROG.CPP
```

This command specifies a precompiled header file named *`MYPCH.pch`*. The compiler uses its contents to restore the precompiled state of all header files up to and including *`MYAPP.h`*. The compiler then compiles the code that occurs after the `#include "MYAPP.h"`* directive.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/Z7`, `/Zi`, `/ZI` (Debug Information Format)

Article • 12/10/2021

The `/Z7`, `/Zi`, and `/ZI` compiler options specify the type of debugging information created for your program, and whether this information is kept in object files or in a program database (PDB) file.

Syntax

> `/Z7`
> `/Zi`
> `/ZI`

Remarks

When you specify a debug option, the compiler produces symbol names for functions and variables, type information, and line locations for use by the debugger. This symbolic debugging information can be included either in the object files (*`.obj`* files) produced by the compiler, or in a separate PDB file (a *`.pdb`* file) for the executable. The debug information format options are described in the following sections.

None

By default, if no debug information format option is specified, the compiler produces no debugging information, so compilation is faster.

`/Z7`

The `/Z7` option produces object files that also contain full symbolic debugging information for use with the debugger. These object files and any libraries built from them can be substantially larger than files that have no debugging information. The symbolic debugging information includes the names and types of variables, functions, and line numbers. No PDB file is produced by the compiler. However, a PDB file can still be generated from these object files or libraries if the linker is passed the `/DEBUG` option.

For distributors of debug versions of third-party libraries, there's an advantage to not having a PDB file. However, the object files for any precompiled headers are necessary

during the library link phase, and for debugging. If there's only type information (and no code) in the *.pch* object file, you must also use the /Yl (Inject PCH Reference for Debug Library) option, which is enabled by default, when you build the library.

The deprecated /Gm (Enable Minimal Rebuild) option is unavailable when **/Z7** is specified.

/Zi

The **/Zi** option produces a separate PDB file that contains all the symbolic debugging information for use with the debugger. The debugging information isn't included in the object files or executable, which makes them much smaller.

Use of **/Zi** doesn't affect optimizations. However, **/Zi** does imply **/debug**. For more information, see /DEBUG (Generate Debug Info).

When you specify both **/Zi** and **/clr**, the DebuggableAttribute attribute isn't placed in the assembly metadata. If you want it, you must specify it in the source code. This attribute can affect the runtime performance of the application. For more information about how the `Debuggable` attribute affects performance and how you can modify the performance impact, see Making an image easier to debug.

The compiler names the PDB file *<project>.pdb*, where *<project>* is the name of your project. If you compile a file outside of a project, the compiler creates a PDB file named *VC<x>.pdb*, where *<x>* is a concatenation of the major and minor version number of the compiler version in use. The compiler embeds the name of the PDB and an identifying timestamped signature in each object file created using this option. This name and signature point the debugger to the location of symbolic and line-number information. The name and signature in the PDB file must match the executable for symbols to be loaded in the debugger. The WinDBG debugger can load mismatched symbols by using the **.symopt+0x40** command. Visual Studio doesn't have a similar option to load mismatched symbols.

If you create a library from objects that were compiled using **/Zi**, the associated PDB file must be available when the library is linked to a program. That means, if you distribute the library, you must also distribute the PDB file. To create a library that contains debugging information without using PDB files, you must select the **/Z7** option. If you use the precompiled headers options, debugging information for both the precompiled header and the rest of the source code is placed in the PDB file.

/ZI

The `/ZI` option is similar to `/Zi`, but it produces a PDB file in a format that supports the Edit and Continue feature. To use Edit and Continue debugging features, you must use this option. The Edit and Continue feature is useful for developer productivity, but can cause issues in code size, performance, and compiler conformance. Because most optimizations are incompatible with Edit and Continue, using `/ZI` disables any `#pragma optimize` statements in your code. The `/ZI` option is also incompatible with use of the __LINE__ predefined macro; code compiled with `/ZI` can't use `__LINE__` as a non-type template argument, although `__LINE__` can be used in macro expansions.

The `/ZI` option forces both the /Gy (Enable Function-Level Linking) and /FC (Full Path of Source Code File in Diagnostics) options to be used in your compilation.

`/ZI` is incompatible with /clr (Common Language Runtime Compilation).

> **Note**
>
> The `/ZI` option is only available in the compilers targeting x86 and x64 processors. This compiler option is not available in the compilers targeting ARM processors.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **General** property page.
3. Modify the **Debug Information Format** property. Choose **OK** to save your changes.

To set this compiler option programmatically

- See DebugInformationFormat.

See also

MSVC compiler options
MSVC compiler command-line syntax

/Za, /Ze (Disable Language Extensions)

Article • 08/03/2021

The **/Za** compiler option disables and emits errors for Microsoft extensions to C that aren't compatible with ANSI C89/ISO C90. The deprecated **/Ze** compiler option enables Microsoft extensions. Microsoft extensions are enabled by default.

Syntax

> **/Za**
> **/Ze**

Remarks

> **Note**
>
> The use of **/Za** when code is compiled as C++ is not recommended. The **/Ze** option is deprecated because its behavior is on by default. For a list of deprecated compiler options, see **Deprecated and removed compiler options**.

The Microsoft C/C++ compiler supports compilation of C code in two ways:

- The compiler uses C compilation mode by default when a source file has a *.c* extension, or when the /Tc or /TC option is specified. The C compiler is an C89/C90 compiler that, by default, enables Microsoft extensions to the C language. For more information about specific extensions, see Microsoft Extensions to C and C++. When both C compilation and the **/Za** option are specified, the C compiler conforms strictly to the C89/C90 standard. The compiler treats Microsoft extended keywords as simple identifiers, disables the other Microsoft extensions, and automatically defines the __STDC__ predefined macro for C programs.

- The compiler can compile C code in C++ compilation mode. This behavior is the default for source files that don't have a *.c* extension, and when the /Tp or /TP option is specified. In C++ compilation mode, the compiler supports those parts of the ISO C99 and C11 standards that have been incorporated into the C++ standard. Almost all C code is also valid C++ code. A small number of C keywords and code constructs aren't valid C++ code, or are interpreted differently in C++. The compiler behaves according to the C++ standard in these cases. In C++

compilation mode, the **/Za** option may cause unexpected behavior and isn't recommended.

Other compiler options can affect how the compiler ensures standards conformance. For ways to specify specific standard C and C++ behavior settings, see the /Zc compiler option. For additional C++ standard conformance settings, see the /permissive- and /std compiler options.

For more information about conformance issues with Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Language** property page.
3. Modify the **Disable Language Extensions** property.

To set this compiler option programmatically

See DisableLanguageExtensions.

See also

Compiler Options
/Zc (Conformance)
/permissive- (Standards conformance)
/std (Specify Language Standard Version)

Microsoft extensions to C and C++

Article • 03/22/2022

Microsoft Visual C++ (MSVC) extends the C and C++ language standards in several ways, detailed in this article.

The MSVC C++ compiler defaults to support for ISO C++14 with some ISO C++17 features and some Microsoft-specific language extensions. For more information on supported features, see Microsoft C/C++ language conformance by Visual Studio version. You can use the `/std` compiler option to enable full support for ISO C++17 and ISO C++20 language features. For more information, see /std (Specify language standard version).

Where specified, some MSVC C++ language extensions can be disabled by use of the `/Za` compiler option. In Visual Studio 2017 and later versions, the `/permissive-` compiler option disables Microsoft-specific C++ language extensions. The `/permissive-` compiler option is implicitly enabled by the `/std:c++20` and `/std:c++latest` compiler options.

By default, when MSVC compiles code as C, it implements ANSI C89 with Microsoft-specific language extensions. Some of these MSVC extensions are standardized in ISO C99 and later. Most MSVC C extensions can be disabled by use of the `/Za` compiler option, as detailed later in this article. You can use the `/std` compiler option to enable support for ISO C11 and C17. For more information, see /std (Specify language standard version).

The standard C runtime library is implemented by the Universal C runtime library (UCRT) in Windows. The UCRT also implements many POSIX and Microsoft-specific library extensions. The UCRT supports the ISO C11 and C17 C runtime library standards, with certain implementation-specific caveats. It doesn't support the full ISO C99 standard C runtime library. For more information, see compatibility in the Universal C runtime library documentation.

Keywords

MSVC adds several Microsoft-specific keywords to C and C++. In the list in Keywords, the keywords that have two leading underscores are MSVC extensions.

Casts

Both the C++ compiler and C compiler support these kinds of non-standard casts:

- The C compiler supports non-standard casts to produce l-values. For example:

```
C
```

```
char *p;
(( int * ) p )++;
// In C with /W4, both by default and under /Ze:
//     warning C4213: nonstandard extension used: cast on l-value
// Under /TP or /Za:
//     error C2105: '++' needs l-value
```

> **Note**
>
> This extension is available in the C language only. You can use the following C standard form in C++ code to modify a pointer as if it's a pointer to a different type.

The preceding example could be rewritten as follows to conform to the C standard.

```
C
```

```
p = ( char * )(( int * )p + 1 );
```

- Both the C and C++ compilers support non-standard casts of a function pointer to a data pointer. For example:

```
C
```

```
int ( * pfunc ) ();
int *pdata;
pdata = ( int * ) pfunc;
/* No diagnostic at any level, whether compiled with default options or
under /Za */
```

Variable-length argument lists

Both C and C++ compilers support a function declarator that specifies a variable number of arguments, followed by a function definition that provides a type instead:

```
C++
```

```
void myfunc( int x, ... );
void myfunc( int x, char * c )
{ }
// In C with /W4, either by default or under /Ze:
//     warning C4212: nonstandard extension used: function declaration used
ellipsis
// In C with /W4, under /Za:
//     warning C4028: formal parameter 2 different from declaration
// In C++, no diagnostic by default or under /Za.
```

Single-line comments

The C compiler supports single-line comments, which are introduced by using two forward slash (`//`) characters:

```
C
```

```
// This is a single-line comment.
```

Single-line comments are a C99 feature. They're unaffected by `/Za` and cause no diagnostic at any level.

Scope

The C compiler supports the following scope-related features.

- Redefinitions of `extern` items as `static`:

  ```
  C
  ```

  ```
  extern int clip();
  static int clip() {}
  // In C and C++ with /W4, either by default or under /Ze:
  //     warning C4211: nonstandard extension used: redefined extern to
  static
  // In C and C++ under /Za:
  //     error C2375: 'clip': redefinition; different linkage
  ```

- Use of benign typedef redefinitions within the same scope:

  ```
  C
  ```

  ```
  typedef int INT;
  typedef int INT; // No diagnostic at any level in C or C++
  ```

- Function declarators have file scope:

```
C

void func1()
{
     extern double func2( double );
     // In C at /W4:  warning C4210: nonstandard extension used:
function given file scope
}
int main( void )
{
     func2( 4 );    //  /Ze passes 4 as type double
}                  //  /Za passes 4 as type int
```

- Use of block-scope variables that are initialized by using non-constant expressions:

```
C

int clip( int );
int bar( int );
int main( void )
{
     int array[2] = { clip( 2 ), bar( 4 ) };
}
int clip( int x )
{
     return x;
}
int bar( int x )
{
     return x;
}
```

Data declarations and definitions

The C compiler supports the following data declaration and definition features.

- Mixed character and string constants in an initializer:

```
C

char arr[6] = {'a', 'b', "cde"};
// In C with /W4, either by default or under /Ze:
//     warning C4207: nonstandard extension used: extended initializer
form
// Under /Za:
//     error C2078: too many initializers
```

- Bit fields that have base types other than `unsigned int` or `signed int`.
- Declarators that don't have a type:

```C
x;
// By default or under /Ze, /Za, /std:c11, and /std:c17, when /W4 is
specified:
//     warning C4431: missing type specifier - int assumed. Note: C no
longer supports default-int
//     warning C4218: nonstandard extension used: must specify at least
a storage class or a type
*/
int main( void )
{
     x = 1;
}
```

- Unsized arrays as the last field in structures and unions:

```C
struct zero
{
     char *c;
     int zarray[];
     // In C with /W4, either by default, under /Ze, /std:c11, and
/std:c17:
     //     warning C4200: nonstandard extension used: zero-sized array
in struct/union
     // Under /Za:
     //     error C2133: 'zarray': unknown size
};
```

- Unnamed (anonymous) structures:

```C
struct
{
     int i;
     char *s;
};
// By default or under /Ze, /std:c11, and /std:c17, when /W4 is
specified:
//     warning C4094: untagged 'struct' declared no symbols
// Under /Za:
//     error C2059: syntax error: 'empty declaration'
```

- Unnamed (anonymous) unions:

```C
union
{
     int i;
     float fl;
};
// By default or under /Ze, /std:c11, and /std:c17, when /W4 is
specified:
//     warning C4094: untagged 'union' declared no symbols
// Under /Za:
//     error C2059: syntax error: 'empty declaration'
```

Intrinsic floating-point functions

Both the x86 C++ compiler and C compiler support inline generation of the `atan`, `atan2`, `cos`, `exp`, `log`, `log10`, `sin`, `sqrt`, and `tan` functions when **`/Oi`** is specified. These intrinsics don't conform to the standard, because they don't set the `errno` variable.

`ISO646.H` not enabled

Under **`/Ze`**, you have to include *`iso646.h`* if you want to use text forms of the following operators:

Operator	Text form
`&&`	`and`
`&=`	`and_eq`
`&`	`bitand`
`\|`	`bitor`
`~`	`compl`
`!`	`not`
`!=`	`not_eq`
`\|\|`	`or`
`\|=`	`or_eq`

Operator	Text form
`^`	`xor`
`^=`	`xor_eq`

These text forms are available as C++ keywords under `/Za` or when `/permissive-` is specified or implied.

See also

/Za, /Ze (Disable language extensions)
MSVC compiler options
MSVC compiler command-line syntax

`/Zc` (Conformance)

Article • 08/30/2023

Use the `/Zc` compiler options to specify standard or Microsoft-specific compiler behavior.

Syntax

> `/Zc:`*option*{,*option* ...}

You may set multiple `/Zc` options separated by commas in a single `/Zc` compiler option. If a `/Zc` option is enabled and disabled in the same command, the option that appears last is used.

Remarks

When Visual Studio has implemented an extension to C or C++ that is incompatible with the standard, you can use a `/Zc` conformance option to specify standard-conforming or Microsoft-specific behavior. For some options, the Microsoft-specific behavior is the default, to prevent large-scale breaking changes to existing code. In other cases, the default is the standard behavior, where improvements in security, performance, or compatibility outweigh the costs of breaking changes. The default setting of each conformance option may change in newer versions of Visual Studio. For more information about each conformance option, see the article for the specific option. The /permissive- compiler option implicitly sets the conformance options that aren't set by default to their conforming settings.

Here are the `/Zc` compiler options:

Option	Behavior
/Zc:__cplusplus[-]	Enable the `__cplusplus` macro to report the supported standard. Off by default.
/Zc:__STDC__	Enable the `__STDC__` macro to report the C standard is supported. Off by default.
/Zc:alignedNew[-]	Enable C++17 over-aligned dynamic allocation. Off by default unless `/std:c++17` or later is specified.
/Zc:auto[-]	Enforce the new Standard C++ meaning for `auto`. On by default.

Option	Behavior
/Zc:char8_t[-]	Enable or disable C++20 native `u8` literal support as `const char8_t`. Off by default unless `/std:c++20` or later is specified.
/Zc:enumTypes[-]	Enable Standard C++ rules for `enum` type deduction. Off by default.
/Zc:externC[-]	Enforce Standard C++ rules for `extern "C"` functions. Off by default unless `/permissive-` is specified.
/Zc:externConstexpr[-]	Enable external linkage for `constexpr` variables. Off by default.
/Zc:forScope[-]	Enforce Standard C++ `for` scoping rules. On by default.
/Zc:gotoScope[-]	Enforce Standard C++ `goto` rules around local variable initialization. Off by default unless `/permissive-` is specified.
/Zc:hiddenFriend[-]	Enforce Standard C++ hidden friend rules. Off by default unless `/permissive-` is specified.
/Zc:implicitNoexcept[-]	Enable implicit `noexcept` on required functions. On by default.
/Zc:inline[-]	Remove unreferenced functions or data if they're COMDAT or have internal linkage only. Off by default.
/Zc:lambda[-]	Enable new lambda processor for conformance-mode syntactic checks in generic lambdas. Off by default unless `/std:c++20` or later is specified.
/Zc:noexceptTypes[-]	Enforce C++17 `noexcept` rules. Off by default unless `/std:c++17` or later is specified.
/Zc:nrvo[-]	Enable optional copy and move elisions. Off by default unless `/O2`, `/permissive-`, or `/std:c++20` or later is specified.
/Zc:preprocessor[-]	Use the new conforming preprocessor. Off by default unless `/std:c11` or later is specified.
/Zc:referenceBinding[-]	A UDT temporary won't bind to a nonconst lvalue reference. Off by default unless `/permissive-` is specified.
/Zc:rvalueCast[-]	Enforce Standard C++ explicit type conversion rules. Off by default unless `/permissive-` is specified.
/Zc:sizedDealloc[-]	Enable C++14 global sized deallocation functions. On by default.
/Zc:strictStrings[-]	Disable string-literal to `char*` or `wchar_t*` conversion. Off by default unless `/permissive-` is specified.
/Zc:static_assert[-]	strict handling of `static_assert`. Off by default unless `/permissive-` is specified.

Option	Behavior
/Zc:templateScope[-]	Enforce Standard C++ template parameter shadowing rules. Off by default.
/Zc:ternary[-]	Enforce conditional operator rules on operand types. Off by default unless `/permissive-` is specified.
/Zc:threadSafeInit[-]	Enable thread-safe local static initialization. On by default.
/Zc:throwingNew[-]	Assume `operator new` throws on failure. Off by default.
/Zc:tlsGuards[-]	Generate runtime checks for TLS variable initialization. On by default.
/Zc:trigraphs[-]	Enable trigraphs (obsolete, off by default).
/Zc:twoPhase-	Use nonconforming template parsing behavior (only applicable when `/permissive-` is specified, which defaults to conforming).
/Zc:wchar_t[-]	`wchar_t` is a native type, not a typedef. On by default.
/Zc:zeroSizeArrayNew[-]	Call member `new`/`delete` for 0-size arrays of objects. On by default.

For more information about conformance issues in MSVC, see Nonstandard behavior.

See also

MSVC compiler options
MSVC compiler command-line syntax

`/Zc:__cplusplus` (Enable updated `__cplusplus` macro)

Article • 08/13/2021

The `/Zc:__cplusplus` compiler option enables the `__cplusplus` preprocessor macro to report an updated value for recent C++ language standards support. By default, Visual Studio always returns the value `199711L` for the `__cplusplus` preprocessor macro.

Syntax

> `/Zc:__cplusplus`[`-`]

Remarks

The `__cplusplus` preprocessor macro is commonly used to report support for a particular version of the C++ standard. Because a lot of existing code appears to depend on the value of this macro matching `199711L`, the compiler doesn't change the value of the macro unless you explicitly opt in by using the `/Zc:__cplusplus` compiler option. The `/Zc:__cplusplus` option is available starting in Visual Studio 2017 version 15.7, and is off by default. In earlier versions of Visual Studio, and by default, or if `/Zc:__cplusplus-` is specified, Visual Studio returns the value `199711L` for the `__cplusplus` preprocessor macro. The /permissive- option doesn't enable `/Zc:__cplusplus`.

When the `/Zc:__cplusplus` option is enabled, the value reported by the `__cplusplus` macro depends on the /std version option setting. This table shows the possible values for the macro:

`/Zc:__cplusplus` option	`/std` option	`__cplusplus` value
`Zc:__cplusplus`	`/std:c++14` (default)	`201402L`
`Zc:__cplusplus`	`/std:c++17`	`201703L`
`Zc:__cplusplus`	`/std:c++20`	`202002L`
`Zc:__cplusplus`	`/std:c++latest`	see text
`Zc:__cplusplus-` (disabled)	Any value	`199711L`

`/Zc:__cplusplus` option	`/std` option	`__cplusplus` value
Not specified	Any value	`199711L`

The compiler doesn't support standards options for C++98, C++03, or C++11. The `/std:c++20` option is available starting in Visual Studio 2019 version 16.11. The value of `__cplusplus` with the `/std:c++latest` option depends on the version of Visual Studio. It's always at least one higher than the highest supported `__cplusplus` standard value supported by your version of Visual Studio.

For finer-grained detection of changes to the compiler toolset, use the _MSC_VER predefined macro. The value of this built-in macro is incremented for every toolset update in Visual Studio 2017 and later versions. The _MSVC_LANG predefined macro reports the standard version whether the `/Zc:__cplusplus` option is enabled or disabled. When `/Zc:__cplusplus` is enabled, `__cplusplus` has the same value as `_MSVC_LANG`.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Add `/Zc:__cplusplus` or `/Zc:__cplusplus-` to the **Additional options:** pane.

See also

/Zc (Conformance)
/std (Specify language standard version)
Predefined macros

`/Zc:__STDC__` (Enable `__STDC__` macro)

Article • 11/08/2022

The `/Zc:__STDC__` compiler option defines the built-in `__STDC__` preprocessor macro as 1 in C code.

Syntax

> `/Zc:__STDC__`

Remarks

The `/Zc:__STDC__` compiler option implements Standard C conforming behavior for the `__STDC__` preprocessor macro, setting it to 1 when compiling C11 and C17 code.

The `/Zc:__STDC__` option is new in Visual Studio 2022 version 17.2. This option is off by default, but can be enabled explicitly when `/std:c11` or `/std:c17` is specified. There's no negative version of the option.

This option is a source breaking change. Due to the behavior of the UCRT, which doesn't expose POSIX functions when `__STDC__` is `1`, it isn't possible to define this macro for C by default without introducing breaking changes to the stable language versions.

Example

C

```
// test__STDC__.c
#include <io.h>
#include <fcntl.h>
#include <stdio.h>

int main() {
#if __STDC__
    int f = _open("file.txt", _O_RDONLY);
    _close(f);
#else
    int f = open("file.txt", O_RDONLY);
    close(f);
#endif
}

/* Command line behavior
```

```
C:\Temp>cl /EHsc /W4 /Zc:__STDC__ test__STDC__.c && test__STDC__

*/
```

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In **Additional options**, add `/Zc:__STDC__`. Choose **OK** or **Apply** to save your changes.

See also

/Zc (Conformance)
/std (Specify language standard version)

`/Zc:alignedNew` (C++17 over-aligned allocation)

Article • 08/17/2021

Enable support for C++17 over-aligned `new`, dynamic memory allocation aligned on boundaries greater than the default for the maximum-sized standard aligned type, `max_align_t`.

Syntax

> **`/Zc:alignedNew`**[**`-`**]

Remarks

The MSVC compiler and library support C++17 standard over-aligned dynamic memory allocation. When the **`/Zc:alignedNew`** option is specified, a dynamic allocation such as `new Example;` respects the alignment of `Example` even when it's greater than `max_align_t`, the largest alignment required for any fundamental type. When the alignment of the allocated type is no more than the alignment guaranteed by the original operator **`new`**, available as the value of the predefined macro `__STDCPP_DEFAULT_NEW_ALIGNMENT__`, the statement `new Example;` results in a call to `::operator new(size_t)` as it did in C++14. When the alignment is greater than `__STDCPP_DEFAULT_NEW_ALIGNMENT__`, the implementation instead obtains the memory by using `::operator new(size_t, align_val_t)`. Similarly, deletion of over-aligned types invokes `::operator delete(void*, align_val_t)` or the sized delete signature `::operator delete(void*, size_t, align_val_t)`.

The **`/Zc:alignedNew`** option is only available when /std:c++17 or later is enabled. Under **`/std:c++17`** or later, **`/Zc:alignedNew`** is enabled by default to conform to the C++ standard. If the only reason you implement operator **`new`** and **`delete`** is to support over-aligned allocations, you may no longer need this code in C++17 or later modes. To turn off this option and revert to the C++14 behavior of **`new`** and **`delete`** when you use **`/std::c++17`** or later, specify **`/Zc:alignedNew-`**. If you implement operator **`new`** and **`delete`** but you're not ready to implement the over-aligned operator **`new`** and **`delete`** overloads that have the `align_val_t` parameter, use the **`/Zc:alignedNew-`** option to prevent the compiler and Standard Library from generating calls to the over-aligned

overloads. The /permissive- option doesn't change the default setting of `/Zc:alignedNew`.

Support for `/Zc:alignedNew` is available starting in Visual Studio 2017 version 15.5.

Example

This sample shows how operator `new` and operator `delete` behave when the `/Zc:alignedNew` option is set.

C++

```
// alignedNew.cpp
// Compile by using: cl /EHsc /std:c++17 /W4 alignedNew.cpp
#include <iostream>
#include <malloc.h>
#include <new>

// "old" unaligned overloads
void* operator new(std::size_t size) {
    auto ptr = malloc(size);
    std::cout << "unaligned new(" << size << ") = " << ptr << '\n';
    return ptr ? ptr : throw std::bad_alloc{};
}

void operator delete(void* ptr, std::size_t size) {
    std::cout << "unaligned sized delete(" << ptr << ", " << size << ")\n";
    free(ptr);
}

void operator delete(void* ptr) {
    std::cout << "unaligned unsized delete(" << ptr << ")\n";
    free(ptr);
}

// "new" over-aligned overloads
void* operator new(std::size_t size, std::align_val_t align) {
    auto ptr = _aligned_malloc(size, static_cast<std::size_t>(align));
    std::cout << "aligned new(" << size << ", " <<
        static_cast<std::size_t>(align) << ") = " << ptr << '\n';
    return ptr ? ptr : throw std::bad_alloc{};
}

void operator delete(void* ptr, std::size_t size, std::align_val_t align) {
    std::cout << "aligned sized delete(" << ptr << ", " << size <<
        ", " << static_cast<std::size_t>(align) << ")\n";
    _aligned_free(ptr);
}

void operator delete(void* ptr, std::align_val_t align) {
    std::cout << "aligned unsized delete(" << ptr <<
```

```
        ", " << static_cast<std::size_t>(align) << ")\n";
    _aligned_free(ptr);
}

struct alignas(256) OverAligned {}; // warning C4324, structure is padded

int main() {
    delete new int;
    delete new OverAligned;
}
```

This output is typical for 32-bit builds. The pointer values vary based on where your application runs in memory.

Output

```
unaligned new(4) = 009FD0D0
unaligned sized delete(009FD0D0, 4)
aligned new(256, 256) = 009FE800
aligned sized delete(009FE800, 256, 256)
```

For information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/Zc:alignedNew` or `/Zc:alignedNew-` and then choose **OK**.

See also

/Zc (Conformance)

`/Zc:auto` (Deduce Variable Type)

Article • 08/03/2021

The `/Zc:auto` compiler option tells the compiler how to use the auto keyword to declare variables. If you specify the default option, `/Zc:auto`, the compiler deduces the type of the declared variable from its initialization expression. If you specify `/Zc:auto-`, the compiler allocates the variable to the automatic storage class.

Syntax

> `/Zc:auto`[`-`]

Remarks

The C++ standard defines an original and a revised meaning for the `auto` keyword. Before Visual Studio 2010, the keyword declares a variable in the automatic storage class; that is, a variable that has a local lifetime. Starting with Visual Studio 2010, the keyword deduces the type of a variable from the declaration's initialization expression. Use the `/Zc:auto` compiler option to tell the compiler to use the revised meaning of the `auto` keyword. The `/Zc:auto` option is on by default. The /permissive- option does not change the default setting of `/Zc:auto`.

The compiler issues an appropriate diagnostic message if your use of the `auto` keyword contradicts the current `/Zc:auto` compiler option. For more information, see auto Keyword. For more information about conformance issues with Visual C++, see Nonstandard Behavior.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Add `/Zc:auto` or `/Zc:auto-` to the **Additional options:** pane.

See also

/Zc (Conformance)

auto Keyword

`/Zc:char8_t` (Enable C++20 char8_t type)

Article • 09/02/2021

The `/Zc:char8_t` compiler option enables C++20 conforming `char8_t` type support. `char8_t` is a character type that's used to represent UTF-8 code units.

Syntax

> `/Zc:char8_t`[`-`]

Remarks

The `/Zc:char8_t` compiler option enables the `char8_t` type keyword as specified in the C++20 standard. It causes the compiler to generate `u8` prefixed character or string literals as `const char8_t` or `const char8_t[N]` types, respectively, instead of as `const char` or `const char[N]` types. In C++17, arrays of `char` may be initialized using `u8` string literals. In C++20, this initialization is ill-formed, and causes compiler error C2440. This behavior can be a source-breaking change. You can revert the compiler to C++14 or C++17 behavior explicitly by specifying `/Zc:char8_t-`.

The `/Zc:char8_t` option is available starting in Visual Studio 2019 version 16.1. It's enabled automatically when you specify /std:c++20 or later (such as `/std:c++latest`). Otherwise, it's off by default.

Example

C++

```
const char* s = u8"Hello"; // Compiles in C++17, Error C2440 in C++20
const char8_t* s = u8"Hello"; // Compiles in C++20 or with /Zc:char8_t
```

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

3. Add `/Zc:char8_t` or `/Zc:char8_t-` to the **Additional options:** pane.

See also

/Zc (Conformance)
/std (Specify language standard version)

`/Zc:checkGwOdr` (Enforce Standard C++ ODR violations under `/Gw`)

Article • 09/04/2023

This switch enforces C++ standards conformance when using /Gw (Optimize global data). When using `/Gw`, certain One Definition Rule (ODR) violations are ignored. This flag ensures that the appropriate errors are raised.

Syntax

> `/Zc:checkGwOdr`[`-`]

Remarks

This switch is off by default.

To see an example of ODR violations that are ignored when using `/Gw`, see Standards conformance improvements to /Gw.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/Zc:checkGwOdr` or `/Zc:checkGwOdr-` and then choose **OK**.

See also

/Zc (Conformance)
One Definition Rule (ODR)
Standards conformance improvements to /Gw

`/Zc:enumTypes` (Enable enum type deduction)

Article • 11/08/2022

The `/Zc:enumTypes` compiler option enables C++ conforming `enum` underlying type and enumerator type deduction.

Syntax

> `/Zc:enumTypes`[`-`]

Remarks

The `/Zc:enumTypes` compiler option implements Standard C++ conforming behavior for deduction of enumeration base types and the types of enumerators.

The `/Zc:enumTypes` option is new in Visual Studio 2022 version 17.4. This option is off by default, and isn't enabled by `/permissive-`. To explicitly disable the option, use `/Zc:enumTypes-`.

When enabled, the `/Zc:enumTypes` option is a potential source and binary breaking change. Some enumeration types change size when the conforming `/Zc:enumTypes` option is enabled. Certain Windows SDK headers include such enumeration definitions.

The C++ Standard requires that the underlying type of an enumeration is large enough to hold all enumerators declared in it. Sufficiently large enumerators can set the underlying type of the `enum` to `unsigned int`, `long long`, or `unsigned long long`. Previously, such enumeration types always had an underlying type of `int` in the Microsoft compiler, regardless of enumerator values.

The C++ Standard also specifies that, within an enumeration definition that has no fixed underlying type, the types of enumerators are determined by their initializers. Or, for the enumerators with no initializer, by the type of the previous enumerator (accounting for overflow). Previously, such enumerators were always given the deduced type of the enumeration, with a placeholder for the underlying type (typically `int`).

In versions of Visual Studio before Visual Studio 2022 version 17.4, the C++ compiler didn't correctly determine the underlying type of an unscoped enumeration with no fixed base type. The compiler also didn't correctly model the types of enumerators. It

could assume an incorrect type in enumerations without a fixed underlying type before the closing brace of the enumeration. Under `/Zc:enumTypes`, the compiler correctly implements the standard behavior.

Example: Underlying type of unscoped `enum` with no fixed type

C++

```
enum Unsigned
{
    A = 0xFFFFFFFF // Value 'A' does not fit in 'int'.
};

// Previously, this static_assert failed. It passes with /Zc:enumTypes.
static_assert(std::is_same_v<std::underlying_type_t<Unsigned>, unsigned
int>);

template <typename T>
void f(T x)
{
}

int main()
{
    // Previously called f<int>, now calls f<unsigned int>.
    f(+A);
}

// Previously, this enum would have an underlying type of `int`,
// but Standard C++ requires this to have a 64-bit underlying type.
// The /Zc:enumTypes option changes the size of this enum from 4 to 8,
// which could impact binary compatibility with code compiled with an
// earlier compiler version, or without the switch.
enum Changed
{
    X = -1,
    Y = 0xFFFFFFFF
};
```

Example: Enumerators within an `enum` definition with no fixed underlying type

C++

```
enum Enum {
    A = 'A',
    B = sizeof(A)
```

```
};

static_assert(B == 1); // previously failed, now succeeds under
/Zc:enumTypes
```

In this example the enumerator `A` should have type `char` prior to the closing brace of the enumeration, so `B` should be initialized using `sizeof(char)`. Before the `/Zc:enumTypes` fix, `A` had enumeration type `Enum` with a deduced underlying type `int`, and `B` was initialized using `sizeof(Enum)`, or 4.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In **Additional options**, add `/Zc:enumTypes` or `/Zc:enumTypes-`. Choose **OK** or **Apply** to save your changes.

See also

/Zc (Conformance)
/std (Specify language standard version)

`/Zc:externC` (Use Standard C++ `extern "C"` rules)

Article • 12/03/2021

The `/Zc:externC` compiler option tells the compiler to conform to the C++ standard and enforce consistent parameter declarations for functions declared as `extern "C"`.

Syntax

> `/Zc:externC`
>
> `/Zc:externC-`

Remarks

The `/Zc:externC` compiler option checks the definitions of functions declared by using `extern "C"`.

The `/Zc:externC` option is available starting in Visual Studio 2019 version 16.3. It's off when the /permissive- option isn't set. In earlier versions of Visual Studio, and by default or if `/Zc:externC-` is specified, Visual Studio is permissive about matching declarations of `extern "C"` functions. The `/permissive-` option enables `/Zc:externC`, so it's on by default in projects that use `/std:c++20` or `/std:c++latest`. The `/Zc:externC` option must come after a `/permissive-` option on the command line.

Mismatched `extern "C"` declarations can cause compiler errors C2116 and C2733. In C++ code, an error can occur if you declare an `extern "C"` function more than once and use different parameter types, even if the types have the same definitions. The `/Zc:externC-` option relaxes this check, and doesn't produce these errors.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Add `/Zc:externC` or `/Zc:externC-` to the **Additional options:** pane.

See also

/Zc (Conformance)

`/Zc:externConstexpr` (Enable extern constexpr variables)

Article • 10/13/2023

The `/Zc:externConstexpr` compiler option tells the compiler to conform to the C++ standard and allow external linkage for `constexpr` variables. By default, Visual Studio always gives a `constexpr` variable internal linkage, even if you specify the `extern` keyword.

Syntax

> `/Zc:externConstexpr`[`-`]

Remarks

The `/Zc:externConstexpr` compiler option causes the compiler to apply external linkage to variables declared by using `extern constexpr`.

In earlier versions of Visual Studio, by default or if `/Zc:externConstexpr-` is specified, Visual Studio applies internal linkage to `constexpr` variables even if the `extern` keyword is used. The `/Zc:externConstexpr` option is available starting in Visual Studio 2017 Update 15.6. and is off by default.

As of Visual Studio 2022 Update 17.6, the `/permissive-` option enables both `/Zc:externConstexpr` and /Zc:lambda. In prior versions, `/permissive-` didn't enable either one.

If a header file contains a variable declared `extern constexpr`, it must be marked __declspec(selectany) in order to merge the duplicate declarations into a single instance in the linked binary. Otherwise you may see linker errors, for example, LNK2005, for violations of the one-definition rule.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

3. Add `/Zc:externConstexpr` or `/Zc:externConstexpr-` to the **Additional options:** pane.

See also

auto Keyword
permissive
/Zc (Conformance)

/Zc:forScope (Force Conformance in for Loop Scope)

Article • 08/03/2021

Used to implement standard C++ behavior for for loops with Microsoft extensions (/Ze).

Syntax

> **/Zc:forScope**[-]

Remarks

Standard behavior is to let a `for` loop's initializer go out of scope after the `for` loop. Under **/Zc:forScope-** and /Ze, the `for` loop's initializer remains in scope until the local scope ends.

The **/Zc:forScope** option is on by default. **/Zc:forScope** is not affected when the /permissive- option is specified.

The **/Zc:forScope-** option is deprecated and will be removed in a future release. Use of **/Zc:forScope-** generates deprecation warning D9035.

The following code compiles under **/Ze** but not under **/Za**:

C++

```
// zc_forScope.cpp
// compile by using: cl /Zc:forScope- /Za zc_forScope.cpp
// C2065, D9035 expected
int main() {
    // Compile by using cl /Zc:forScope- zc_forScope.cpp
    // to compile this non-standard code as-is.
    // Uncomment the following line to resolve C2065 for /Za.
    // int i;
    for (int i = 0; i < 1; i++)
        ;
    i = 20;   // i has already gone out of scope under /Za
}
```

If you use **/Zc:forScope-**, warning C4288 (off by default) is generated if a variable is in scope because of a declaration that was made in a previous scope. To demonstrate this, remove the `//` characters in the example code to declare `int i`.

You can modify the run-time behavior of **/Zc:forScope** by using the conform pragma.

If you use **/Zc:forScope-** in a project that has an existing .pch file, a warning is generated, **/Zc:forScope-** is ignored, and compilation continues by using the existing .pch files. If you want a new .pch file generated, use /Yc (Create Precompiled Header File).

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Language** property page.
3. Modify the **Force Conformance in For Loop Scope** property.

To set this compiler option programmatically

- See ForceConformanceInForLoopScope.

See also

/Zc (Conformance)
/Za, /Ze (Disable Language Extensions)

`/Zc:gotoScope` (Enforce conformance in goto scope)

Article • 11/15/2022

The `/Zc:gotoScope` compiler option enables checks for Standard C++ behavior around `goto` statements that jump over the initialization of local variables.

Syntax

> `/Zc:gotoScope`[`-`]

Remarks

The `/Zc:gotoScope` compiler option enforces C++ Standard behavior around `goto` statements that jump over the initialization of one or more local variables. The compiler emits error C2362 in all such cases when `/Zc:gotoScope` is specified. The `/Zc:gotoScope-` relaxes this check, but the compiler still emits an error if a `goto` skips initialization of a local variable that has a non-trivial destructor.

The intent of the `/Zc:gotoScope-` option is to help ease the migration of older code bases to more conformant code. You may use it to suppress certain errors until you've updated the non-conforming code.

The `/Zc:gotoScope` compiler option is new in Visual Studio 2022 version 17.4. The option is off by default. It's enabled automatically by the `/permissive-` option (or an option that implies `/permissive-`, such as `/std:c++20` or `/std:c++latest`). To enable the error check explicitly, add `/Zc:gotoScope` to the compiler command line. To explicitly disable the check, use the `/Zc:gotoScope-` option. The `/Zc:gotoScope-` must appear after the `/permissive-` option or any option that implies `/permissive-`.

Example

This sample generates an error message when compiled using `/Zc:gotoScope`:

C++

```
int g(int*);
bool failed(int);
```

```
int f() {
    int v1;
    auto result = g(&v1);
    if (failed(result))
        goto OnError;
    int v2 = v1 + 2;
    return v2;
OnError:
    return -1;
}

/* Output:
t.cpp(9): error C2362: initialization of 'v2' is skipped by 'goto OnError'
*/
```

If the code is compiled with `/Zc:gotoScope-`, the compiler doesn't emit the error.

Even when `/Zc:gotoScope-` is specified, the compiler still emits an error if the local variable has a non-trivial destructor. For example:

C++

```
int g(int*);
bool failed(int);

class S {
public:
    S(int);
    ~S();
    int mf() const;
};

int f()
{
    int v1;
    auto result = g(&v1);
    if (failed(result))
        goto OnError;
    S s(v1);
    return s.mf();

OnError:
    return -1;
}

/* Output:
t.cpp(17): error C2362: initialization of 's' is skipped by 'goto OnError'
*/
```

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In **Additional options**, add `/Zc:gotoScope` or `/Zc:gotoScope-`. Choose **OK** or **Apply** to save your changes.

See also

/Zc (Conformance)
/permissive-
/std (Specify language standard version)

`/Zc:hiddenFriend` (Enforce Standard C++ hidden friend rules)

Article • 08/03/2021

Specifies the compiler conforms to the C++ standard treatment of hidden friend functions or function templates.

Syntax

> `/Zc:hiddenFriend[-]`

Remarks

The `/Zc:hiddenFriend` option enables a subset of the /permissive- option behavior. It tells the compiler to conform to the standard for hidden friends. The compiler only includes hidden friends in argument-dependent lookup (ADL) for explicit instances or template parameters of the enclosing class type. The restriction allows you to use hidden friends to keep operations on a type from applying to implicit conversions. This option can improve build speed in code that can't otherwise use /permissive-.

A *hidden friend* is a `friend` function or function template declared only within a class or class template definition. By default, the Microsoft C++ compiler doesn't remove hidden friend declarations as candidates for overload resolution everywhere it should. This legacy behavior can slow the compiler down by including the hidden friend functions as possible candidates in more contexts.

Standard C++ hidden friend behavior is enabled by default under `/permissive-`. To specify legacy hidden friend behavior when the `/permissive-` option is specified, use `/Zc:hiddenFriend-`. Use of C++20 Modules requires standard hidden friend behavior.

The `/Zc:hiddenFriend` option is available starting in Visual Studio 2019 version 16.4.

For examples of compiler behavior when you specify `/Zc:hiddenFriend`, see Hidden friend name lookup rules.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/Zc:hiddenFriend` or `/Zc:hiddenFriend-` and then choose **OK**.

See also

/Zc (Conformance)
/permissive-

/Zc:implicitNoexcept (Implicit Exception Specifiers)

Article • 08/03/2021

When the **/Zc:implicitNoexcept** option is specified, the compiler adds an implicit noexcept exception specifier to compiler-defined special member functions and to user-defined destructors and deallocators. By default, **/Zc:implicitNoexcept** is enabled to conform to the ISO C++11 standard. Turning this option off disables implicit `noexcept` on user-defined destructors and dealloacators and compiler-defined special member functions.

Syntax

> **/Zc:implicitNoexcept**[-]

Remarks

/Zc:implicitNoexcept tells the compiler to follow section 15.4 of the ISO C++11 standard. It implicitly adds a `noexcept` exception specifier to each implicitly-declared or explicitly defaulted special member function—the default constructor, copy constructor, move constructor, destructor, copy assignment operator, or move assignment operator—and each user-defined destructor or deallocator function. A user-defined deallocator has an implicit `noexcept(true)` exception specifier. For user-defined destructors, the implicit exception specifier is `noexcept(true)` unless a contained member class or base class has a destructor that is not `noexcept(true)`. For compiler-generated special member functions, if any function directly invoked by this function is effectively `noexcept(false)`, the implicit exception specifier is `noexcept(false)`. Otherwise, the implicit exception specifier is `noexcept(true)`.

The compiler does not generate an implicit exception specifier for functions declared by using explicit `noexcept` or `throw` specifiers or a `__declspec(nothrow)` attribute.

By default, **/Zc:implicitNoexcept** is enabled. The /permissive- option does not affect **/Zc:implicitNoexcept**.

If the option is disabled by specifying **/Zc:implicitNoexcept-**, no implicit exception specifiers are generated by the compiler. This behavior is the same as Visual Studio 2013, where destructors and deallocators that did not have exception specifiers could

have `throw` statements. By default, and when **/Zc:implicitNoexcept** is specified, if a `throw` statement is encountered at run time in a function with an implicit `noexcept(true)` specifier, it causes an immediate invocation of `std::terminate`, and normal unwinding behavior for exception handlers is not guaranteed. To help identify this situation, the compiler generates Compiler Warning (level 1) C4297. If the `throw` is intentional, we recommend you change your function declaration to have an explicit `noexcept(false)` specifier instead of using **/Zc:implicitNoexcept-**.

This sample shows how a user-defined destructor that has no explicit exception specifier behaves when the **/Zc:implicitNoexcept** option is set or disabled. To show the behavior when set, compile by using `cl /EHsc /W4 implicitNoexcept.cpp`. To show the behavior when disabled, compile by using `cl /EHsc /W4 /Zc:implicitNoexcept- implicitNoexcept.cpp`.

C++

```
// implicitNoexcept.cpp
// Compile by using: cl /EHsc /W4 implicitNoexcept.cpp
// Compile by using: cl /EHsc /W4 /Zc:implicitNoexcept- implicitNoexcept.cpp

#include <iostream>
#include <cstdlib>      // for std::exit, EXIT_FAILURE, EXIT_SUCCESS
#include <exception>    // for std::set_terminate

void my_terminate()
{
    std::cout << "Unexpected throw caused std::terminate" << std::endl;
    std::cout << "Exit returning EXIT_FAILURE" << std::endl;
    std::exit(EXIT_FAILURE);
}

struct A {
    // Explicit noexcept overrides implicit exception specification
    ~A() noexcept(false) {
        throw 1;
    }
};

struct B : public A {
    // Compiler-generated ~B() definition inherits noexcept(false)
    ~B() = default;
};

struct C {
    // By default, the compiler generates an implicit noexcept(true)
    // specifier for this user-defined destructor. To enable it to
    // throw an exception, use an explicit noexcept(false) specifier,
    // or compile by using /Zc:implicitNoexcept-
    ~C() {
```

```
        throw 1; // C4297, calls std::terminate() at run time
    }
};

struct D : public C {
    // This destructor gets the implicit specifier of its base.
    ~D() = default;
};

int main()
{
    std::set_terminate(my_terminate);

    try
    {
        {
            B b;
        }
    }
    catch (...)
    {
        // exception should reach here in all cases
        std::cout << "~B Exception caught" << std::endl;
    }
    try
    {
        {
            D d;
        }
    }
    catch (...)
    {
        // exception should not reach here if /Zc:implicitNoexcept
        std::cout << "~D Exception caught" << std::endl;
    }
    std::cout << "Exit returning EXIT_SUCCESS" << std::endl;
    return EXIT_SUCCESS;
}
```

When compiled by using the default setting **/Zc:implicitNoexcept**, the sample generates this output:

Output

```
~B Exception caught
Unexpected throw caused std::terminate
Exit returning EXIT_FAILURE
```

When compiled by using the setting **/Zc:implicitNoexcept-**, the sample generates this output:

Output

```
~B Exception caught
~D Exception caught
Exit returning EXIT_SUCCESS
```

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include **/Zc:implicitNoexcept** or **/Zc:implicitNoexcept-** and then choose **OK**.

See also

/Zc (Conformance)
noexcept
Exception Specifications (throw)
terminate

`/Zc:inline` (Remove unreferenced COMDAT)

Article • 06/30/2022

Removes unreferenced data or functions that are COMDATs, or that only have internal linkage. Under `/Zc:inline`, the compiler specifies that translation units with inline data or functions must also include their definitions.

Syntax

> `/Zc:inline`[`-`]

Remarks

When `/Zc:inline` is specified, the compiler doesn't emit symbol information for unreferenced COMDAT functions or data. Or, for data or functions that have internal linkage only. This optimization simplifies some of the work the linker does in release builds, or when you specify the /OPT:REF linker option. This compiler optimization can significantly reduce .obj file size and improve linker speeds. The compiler option isn't enabled when you disable optimizations (/Od). Or, when you specify /GL (Whole Program Optimization).

By default, this option is off (`/Zc:inline-`) in command-line builds. The /permissive- option doesn't enable `/Zc:inline`. In MSBuild projects, the option is set by the **Configuration Properties** > **C/C++** > **Language** > **Remove unreferenced code and data** property, which is set to **Yes** by default.

If `/Zc:inline` is specified, the compiler enforces the C++11 requirement that all functions declared `inline` must have a definition available in the same translation unit if they're used. When the option isn't specified, the Microsoft compiler allows non-conformant code that invokes functions declared `inline` even if no definition is visible. For more information, see the C++11 standard, in section 3.2 and section 7.1.2. This compiler option was introduced in Visual Studio 2013 Update 2.

To use the `/Zc:inline` option, update non-conforming code.

This example shows how the non-conforming use of an inline function declaration without a definition still compiles and links when the default `/Zc:inline-` option is used:

Source file `example.h`:

```C++
// example.h
// Compile by using: cl /W4 /EHsc /O2 zcinline.cpp example.cpp
#pragma once

class Example {
public:
   inline void inline_call(); // declared but not defined inline
   void normal_call();
   Example() {};
};
```

Source file `example.cpp`:

```C++
// example.cpp
// Compile by using: cl /W4 /EHsc /O2 zcinline.cpp example.cpp
#include <stdio.h>
#include "example.h"

void Example::inline_call() {
   printf("inline_call was called.\n");
}

void Example::normal_call() {
   printf("normal_call was called.\n");
   inline_call(); // with /Zc:inline-, inline_call forced into .obj file
}
```

Source file `zcinline.cpp`:

```C++
// zcinline.cpp
// Compile by using: cl /W4 /EHsc /O2 zcinline.cpp example.cpp
#include "example.h"

int main() {
   Example example;
   example.inline_call(); // normal call when definition unavailable
}
```

When **`/Zc:inline`** is enabled, the same code causes a LNK2019 error, because the compiler doesn't emit a non-inlined code body for `Example::inline_call` in

`example.obj`. The missing code causes the non-inlined call in `main` to reference an undefined external symbol.

To resolve this error, you can remove the **`inline`** keyword from the declaration of `Example::inline_call`, or move the definition of `Example::inline_call` into the header file, or move the implementation of `Example` into `main.cpp`. The next example moves the definition into the header file, where it's visible to any caller that includes the header.

Source file `example2.h`:

C++

```
// example2.h
// Compile by using: cl /W4 /EHsc /O2 zcinline2.cpp example2.cpp
#pragma once
#include <stdio.h>

class Example2 {
public:
   inline void inline_call() {
      printf("inline_call was called.\n");
   }
   void normal_call();
   Example2() {};
};
```

Source file `example2.cpp`:

C++

```
// example2.cpp
// Compile by using: cl /W4 /EHsc /O2 zcinline2.cpp example2.cpp
#include "example2.h"

void Example2::normal_call() {
   printf("normal_call was called.\n");
   inline_call();
}
```

Source file `zcinline2.h`:

C++

```
// zcinline2.cpp
// Compile by using: cl /W4 /EHsc /O2 zcinline2.cpp example2.cpp
#include "example2.h"

int main() {
```

```
    Example2 example2;
    example2.inline_call(); // normal call when definition unavailable
}
```

For more information about conformance issues in Visual C++, see Nonstandard behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Language** property page.
3. Modify the **Remove unreferenced code and data** property, and then choose **OK**.

See also

/Zc (Conformance)

`/Zc:lambda` (Enable updated lambda processor)

Article • 02/19/2025

The `/Zc:lambda` compiler option enables conforming lambda grammar and processing support.

Syntax

> `/Zc:lambda`[`-`]

Remarks

The `/Zc:lambda` compiler option enables the conforming lambda processor. It parses and implements lambda code according to the C++ standard. This option is off by default, which uses the legacy lambda processor. Use this option to enable conformance-mode syntax checks of generic lambdas when you use the default /std:c++14 or the /std:c++17 compiler options.

`/Zc:lambda` is automatically enabled by the /std:c++20, /std:c++latest, /permissive-, and /experimental:module options. You can disable it explicitly by using `/Zc:lambda-`.

The `/Zc:lambda` option is available starting in Visual Studio 2019 version 16.8. It's available as `/experimental:newLambdaProcessor` starting in Visual Studio 2019 version 16.3, but this spelling is now deprecated.

The legacy lambda processor has limitations when it parses and compiles lambdas. For example, this conforming code compiles correctly under `/Zc:lambda`, but reports errors under `/Zc:lambda-`:

C++

```
void f1()
{
    constexpr auto c_value = 1;
    auto func = []()
    {
        return c_value; // error C3493: 'c_value' cannot be implicitly
captured
                        // because no default capture mode has been
specified
```

```
    };
    func(); // error C2064: term does not evaluate to a function taking 0
arguments
}
```

The legacy lambda processor compiles this code without warnings, but the new lambda processor produces error C2760:

C++

```
void f2() {
    auto a = [](auto arg) {
        decltype(arg)::Type t; // C2760 syntax error: unexpected token
'identifier', expected ';'
    };
}
```

This example shows the correct syntax, now enforced by the compiler under `/Zc:lambda`:

C++

```
void f3() {
    auto a = [](auto arg) {
        typename decltype(arg)::Type t;
    };
}
```

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Add `/Zc:lambda` or `/Zc:lambda-` to the **Additional options:** pane.

See also

/Zc (Conformance)
/std (Specify language standard version)

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/Zc:noexceptTypes` (C++17 noexcept rules)

Article • 08/17/2021

The C++17 standard makes `throw()` an alias for **`noexcept`**, removes `throw(` *`type-list`* `)` and `throw(...)`, and allows certain types to include **`noexcept`**. This change can cause a number of source compatibility issues in code that conforms to C++14 or earlier. The **`/Zc:noexceptTypes`** option specifies conformance to the C++17 standard. **`/Zc:noexceptTypes-`** allows the C++14 and earlier behavior when code is compiled in C++17 mode.

Syntax

> **`/Zc:noexceptTypes`**[**`-`**]

Remarks

When the **`/Zc:noexceptTypes`** option is specified, the compiler conforms to the C++17 standard and treats throw() as an alias for noexcept, removes `throw(` *`type-list`* `)` and `throw(...)`, and allows certain types to include **`noexcept`**. The **`/Zc:noexceptTypes`** option is only available when /std:c++17 or later is enabled. **`/Zc:noexceptTypes`** is enabled by default to conform to the ISO C++17 and later standards. The /permissive- option doesn't affect **`/Zc:noexceptTypes`**. Turn off this option by specifying **`/Zc:noexceptTypes-`** to revert to the C++14 behavior of **`noexcept`** when **`/std:c++17`** or later is specified.

Beginning in Visual Studio 2017 version 15.5, the C++ compiler diagnoses more mismatched exception specifications in declarations in C++17 mode, or when you specify the /permissive- option.

This sample shows how declarations with an exception specifier behave when the **`/Zc:noexceptTypes`** option is set or disabled. To show the behavior when set, compile by using `cl /EHsc /W4 noexceptTypes.cpp`. To show the behavior when disabled, compile by using `cl /EHsc /W4 /Zc:noexceptTypes- noexceptTypes.cpp`.

C++

```
// noexceptTypes.cpp
// Compile by using: cl /EHsc /W4 noexceptTypes.cpp
// Compile by using: cl /EHsc /W4 /Zc:noexceptTypes- noexceptTypes.cpp
```

```
void f() throw();    // equivalent to void f() noexcept;
void f() { }         // warning C5043
void g() throw(...); // warning C5040

struct A
{
    virtual void f() throw();
};

struct B : A
{
    virtual void f() { } // error C2694
};
```

When compiled by using the default setting `/Zc:noexceptTypes`, the sample generates the listed warnings. To update your code, use the following instead:

C++

```
void f() noexcept;
void f() noexcept { }
void g() noexcept(false);

struct A
{
    virtual void f() noexcept;
};

struct B : A
{
    virtual void f() noexcept { }
};
```

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/Zc:noexceptTypes` or `/Zc:noexceptTypes-` and then choose **OK**.

See also

/Zc (Conformance)

noexcept

Exception specifications (throw)

`/Zc:nrvo` (Control optional NRVO)

Article • 11/15/2022

The `/Zc:nrvo` compiler option controls Standard C++ optional named return value optimization (NRVO) copy or move elision behavior.

Syntax

> `/Zc:nrvo`[`-`]

Remarks

In Visual Studio 2022 version 17.4 and later, you can explicitly enable optional copy or move elision behavior by using the `/Zc:nrvo` compiler option. This option is off by default, but is set automatically when you compile using the `/O2` option, the `/permissive-` option, or `/std:c++20` or later. Under `/Zc:nrvo`, copy and move elision is performed wherever possible. Optional copy or move elision can also be explicitly disabled by using the `/Zc:nrvo-` option. These compiler options only control optional copy or move elision. Mandatory copy or move elision (specified by the C++ Standard) can't be disabled.

Mandatory copy and move elision

The C++ standard requires copy or move elision when the returned value is initialized as part of the return statement. For example, it's required when a function returns an `ExampleType` returned by using `return ExampleType();`. The MSVC compiler always performs copy and move elision for `return` statements when it's required, even under `/Zc:nrvo-`.

Optional copy and move elision

When a `return` statement contains an expression of non-primitive type, its execution copies the expression result into the return slot of the calling function. The compiler invokes the copy or move constructor of the returned type. Then, as the function is exited, destructors for function-local variables are called, which includes any variables named in the expression.

The C++ standard allows (but doesn't require) the compiler to optionally construct the returned object directly in the return slot of the calling function. This construction skips (or *elides*) the copy or move constructor executed as part of the `return` statement. Unlike most other optimizations, this transformation is allowed to have an observable effect on the program's output. Namely, the copy or move constructor and associated destructor are called one less time. The standard still requires that the named returned variable has a defined copy or move constructor, even if the compiler elides the constructor in all cases.

In versions before Visual Studio 2022 version 17.4, when optimizations are disabled (such as under `/Od` or in functions marked `#pragma optimize("", off)`) the compiler only performs mandatory copy and move elision. Under `/O2`, the older compilers perform optional copy or move elision on return of a named variable in an optimized function when all of these conditions are met: it has no loops or exception handling, it doesn't return multiple symbols with overlapping lifetimes, the type's copy or move constructor doesn't have default arguments.

Visual Studio 2022 version 17.4 increases the number of places where the compiler does optional copy or move elisions under `/Zc:nrvo`, whether enabled explicitly, or automatically by using the `/O2`, `/permissive-`, or `/std:c++20` or later options. Under `/Zc:nrvo`, the compiler performs optional copy or move elision on return of a named variable for any function when: it has no loops or exception handling (as before); it returns the variable from a loop; it has exception handling; the returned type's copy or move constructor has default arguments. Optional copy or move elisions are never done when `/Zc:nrvo-` is applied, or when the function returns multiple symbols with overlapping lifetimes, or for a throw of a named variable.

For more information and examples of mandatory and optional copy elision under `/Zc:nrvo`, see Improving Copy and Move Elision in the C++ Team Blog.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In **Additional options**, add `/Zc:nrvo` or `/Zc:nrvo-`. Choose **OK** or **Apply** to save your changes.

See also

/Zc (Conformance)
/O2
/permissive-
/std (Specify language standard version)

`/Zc:preprocessor` (Enable preprocessor conformance mode)

Article • 02/18/2022

This option enables a token-based preprocessor that conforms to C99 and C++11 and later standards. For more information, see MSVC new preprocessor overview.

Syntax

> `/Zc:preprocessor`[`-`]

Remarks

Use the `/Zc:preprocessor` compiler option to enable the conforming preprocessor. You can use `/Zc:preprocessor-` option to explicitly specify the traditional (non-conforming) preprocessor.

The `/Zc:preprocessor` option is available starting in Visual Studio 2019 version 16.5. An earlier, incomplete version of the new preprocessor option is available in versions of Visual Studio starting in Visual Studio 2017 version 15.8. For more information, see /experimental:preprocessor.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Preprocessor** property page.
3. Modify the **Use Standard Conforming Preprocessor** property and then choose **OK**.

See also

/Zc (Conformance)

/Zc:referenceBinding (Enforce reference binding rules)

Article • 08/03/2021

When the **/Zc:referenceBinding** option is specified, the compiler doesn't allow a non-const lvalue reference to bind to a temporary.

Syntax

> **/Zc:referenceBinding**[-]

Remarks

If **/Zc:referenceBinding** is specified, the compiler follows section 8.5.3 of the C++11 standard: It doesn't allow expressions that bind a user-defined type temporary to a non-const lvalue reference. By default, or if **/Zc:referenceBinding-** is specified, the compiler allows such expressions as a Microsoft extension, but a level 4 warning is issued. For code security, portability and conformance, we recommend you use **/Zc:referenceBinding**.

The **/Zc:referenceBinding** option is off by default. The /permissive- compiler option implicitly sets this option, but it can be overridden by using **/Zc:referenceBinding-**.

Example

This sample shows the Microsoft extension that allows a temporary of a user-defined type to be bound to a non-const lvalue reference.

C++

```
// zcreferencebinding.cpp
struct S {
};

void f(S&) {
}

S g() {
    return S{};
}

int main() {
```

```
    S& s = g();          // warning C4239 at /W4
    const S& cs = g();   // okay, bound to const ref
    f(g());              // Extension: error C2664 only if
/Zc:referenceBinding
}
```

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include **/Zc:referenceBinding** and then choose **OK**.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
/Zc (Conformance)

/Zc:rvalueCast (Enforce type conversion rules)

Article • 08/03/2021

When the `/Zc:rvalueCast` option is specified, the compiler correctly identifies an rvalue reference type as the result of a cast operation. Its behavior conforms to the C++11 standard. When the option is unspecified, the compiler behavior is the same as in Visual Studio 2012.

Syntax

> `/Zc:rvalueCast`
>
> `/Zc:rvalueCast-`

Remarks

If `/Zc:rvalueCast` is specified, the compiler follows section 5.4 of the C++11 standard and treats only cast expressions that result in non-reference types and cast expressions that result in rvalue references to non-function types as rvalue types. By default, or if `/Zc:rvalueCast-` is specified, the compiler is non-conforming, and treats all cast expressions that result in rvalue references as rvalues. For conformance, and to eliminate errors in the use of casts, we recommend that you use `/Zc:rvalueCast`.

By default, `/Zc:rvalueCast` is off (`/Zc:rvalueCast-`). The /permissive- compiler option implicitly sets this option, but it can be overridden by using `/Zc:rvalueCast-`.

Use `/Zc:rvalueCast` if you pass a cast expression as an argument to a function that takes an rvalue reference type. The default behavior causes compiler error C2664 when the compiler incorrectly determines the type of the cast expression. This example shows a compiler error in correct code when `/Zc:rvalueCast` isn't specified:

C++

```
// Test of /Zc:rvalueCast
// compile by using:
// cl /c /Zc:rvalueCast- make_thing.cpp
// cl /c /Zc:rvalueCast make_thing.cpp

#include <utility>
```

```
template <typename T>
struct Thing {
   // Construct a Thing by using two rvalue reference parameters
   Thing(T&& t1, T&& t2)
      : thing1(t1), thing2(t2) {}

   T& thing1;
   T& thing2;
};

// Create a Thing, using move semantics if possible
template <typename T>
Thing<T> make_thing(T&& t1, T&& t2)
{
   return (Thing<T>(std::forward<T>(t1), std::forward<T>(t2)));
}

struct Test1 {
   long a;
   long b;

   Thing<long> test() {
      // Use identity casts to create rvalues as arguments
      return make_thing(static_cast<long>(a), static_cast<long>(b));
   }
};
```

The default compiler behavior may not report error C2102 when appropriate. In this example, the compiler doesn't report an error if the address of an rvalue created by an identity cast is taken when `/Zc:rvalueCast` is unspecified:

C++

```
int main() {
   int a = 1;
   int *p = &a;   // Okay, take address of lvalue
                  // Identity cast creates rvalue from lvalue;
   p = &(int)a;   // problem: should cause C2102: '&' requires l-value
}
```

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Language** property page.

3. Set the **Enforce type conversion rules** property to `/Zc:rvalueCast` or `/Zc:rvalueCast-`. Choose **OK** or **Apply** to save your changes.

See also

/Zc (Conformance)

/Zc:sizedDealloc (Enable Global Sized Deallocation Functions)

Article • 08/03/2021

The **/Zc:sizedDealloc** compiler option tells the compiler to preferentially call global `operator delete` or `operator delete[]` functions that have a second parameter of type `size_t` when the size of the object is available. These functions may use the `size_t` parameter to optimize deallocator performance.

Syntax

> **/Zc:sizedDealloc**[-]

Remarks

In the C++11 standard, you may define static member functions `operator delete` and `operator delete[]` that take a second, `size_t` parameter. Typically these are used in combination with operator new functions to implement more efficient allocators and deallocators for the object. However, C++11 did not define an equivalent set of deallocation functions at global scope. In C++11, global deallocation functions that have a second parameter of type `size_t` are considered placement delete functions. They must be explicitly called by passing a size argument.

The C++14 standard changes the behavior of the compiler. When you define global `operator delete` and `operator delete[]` that take a second parameter of type `size_t`, the compiler prefers to call these functions when member scope versions are not invoked and the size of the object is available. The compiler passes the size argument implicitly. The single argument versions are called when the compiler can't determine the size of the object being deallocated. Otherwise, the usual rules for choosing the version of the deallocation function to invoke still apply. Calls to the global functions may be explicitly specified by prepending the scope resolution operator (`::`) to the deallocation function call.

By default, Visual C++ starting in Visual Studio 2015 implements this C++14 standard behavior. You may explicitly specify this by setting the **/Zc:sizedDealloc** compiler option. This represents a potentially breaking change. Use the **/Zc:sizedDealloc-** option to preserve the old behavior, for example, when your code defines placement delete operators that use a second parameter of type `size_t`. The default Visual Studio library

implementations of the global deallocation functions that have the second parameter of type `size_t` invoke the single parameter versions. If your code supplies only single-parameter global operator delete and operator delete[], the default library implementations of the global sized deallocation functions invoke your global functions.

The **/Zc:sizedDealloc** compiler option is on by default. The /permissive- option does not affect **/Zc:sizedDealloc**.

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. From the **Configurations** drop down menu, choose **All Configurations**.
3. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
4. Modify the **Additional Options** property to include **/Zc:sizedDealloc** or **/Zc:sizedDealloc-** and then choose **OK**.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
/Zc (Conformance)

`/Zc:static_assert` (Strict static_assert handling)

Article • 01/17/2025

The `/Zc:static_assert` compiler option tells the compiler to evaluate `static_assert` calls with non-dependent test expressions when class or function templates are parsed.

Syntax

> `/Zc:static_assert`
> `/Zc:static_assert-`

Remarks

Starting with Visual Studio 17.10, `/Zc:static_assert` and `/Zc:static_assert-` have no effect. Both options are ignored to avoid breaking builds that use them. `static_assert` is now never evaluated when parsing class or function templates.

The `/Zc:static_assert` compiler option tells the compiler to evaluate a `static_assert` in the body of a function template or in the body of a class template member function when first parsed, if the test expression isn't dependent. If the non-dependent test expression is `false`, the compiler emits an error immediately. When the test expression is dependent, the `static_assert` isn't evaluated until the template is instantiated.

The `/Zc:static_assert` option is available starting in Visual Studio 2022 version 17.1. In earlier versions of Visual Studio, or if `/Zc:static_assert-` is specified, Visual Studio doesn't do dependent analysis if the `static_assert` is within the body of a function template or within the body of a member function of a class template. Instead, it only evaluates the `static_assert` when a template is instantiated.

The `/permissive-` option enables `/Zc:static_assert`, so it's on by default in projects that use `/std:c++20` or `/std:c++latest`. The `/Zc:static_assert-` option must come after a `/std:c++20`, `/std:c++latest`, or `/permissive-` option on the command line.

If the compiler is in the default C++14 mode and `/permissive-` or `/Zc:static_assert` is specified, it uses `/Zc:static_assert` behavior. However, if it evaluates a `static_assert` in a template body, it also reports off-by-default warning C5254, "language feature

'terse static assert' requires compiler flag '`/std:c++17`'", since this behavior isn't required until C++17.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Add `/Zc:static_assert` or `/Zc:static_assert-` to the **Additional options:** pane.

See also

/Zc (Conformance)

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`/Zc:strictStrings` (Disable string literal type conversion)

Article • 08/03/2021

When specified, the compiler requires strict `const`-qualification conformance for pointers initialized by using string literals.

Syntax

> `/Zc:strictStrings`[`-`]

Remarks

If `/Zc:strictStrings` is specified, the compiler enforces the standard C++ `const` qualifications for string literals, as type 'array of `const char`' or 'array of `const wchar_t`', depending on the declaration. String literals are immutable, and an attempt to modify the contents of one results in an access violation error at run time. You must declare a string pointer as `const` to initialize it by using a string literal, or use an explicit `const_cast` to initialize a non-`const` pointer. By default, or if `/Zc:strictStrings-` is specified, the compiler does not enforce the standard C++ `const` qualifications for string pointers initialized by using string literals.

The `/Zc:strictStrings` option is off by default. The /permissive- compiler option implicitly sets this option, but it can be overridden by using `/Zc:strictStrings-`.

Use the `/Zc:strictStrings` option to prevent compilation of incorrect code. This example shows how a simple declaration error leads to a crash at run time:

C++

```
// strictStrings_off.cpp
// compile by using: cl /W4 strictStrings_off.cpp
int main() {
   wchar_t* str = L"hello";
   str[2] = L'a'; // run-time error: access violation
}
```

When `/Zc:strictStrings` is enabled, the same code reports an error in the declaration of `str`.

C++

```
// strictStrings_on.cpp
// compile by using: cl /Zc:strictStrings /W4 strictStrings_on.cpp
int main() {
   wchar_t* str = L"hello"; // error: Conversion from string literal
   // loses const qualifier
   str[2] = L'a';
}
```

If you use `auto` to declare a string pointer, the compiler creates the correct `const` pointer type declaration for you. An attempt to modify the contents of a `const` pointer is reported by the compiler as an error.

> **Note**
>
> The C++ Standard Library in Visual Studio 2013 does not support the `/Zc:strictStrings` compiler option in debug builds. If you see several **C2665** errors in your build output, this may be the cause.

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/Zc:strictStrings` and then choose **OK**.

See also

/Zc (Conformance)

`/Zc:templateScope` (Check template parameter shadowing)

Article • 11/15/2022

The `/Zc:templateScope` compiler option enables checks for Standard C++ behavior around shadowing of template parameters.

Syntax

> `/Zc:templateScope`[`-`]

Remarks

The C++ Standard doesn't allow the reuse of a template parameter's name (or *shadowing*) for another declaration within the scope of the template. The `/Zc:templateScope` compiler option enables an error check for such shadowing.

The `/Zc:templateScope` option is new in Visual Studio 2022 version 17.5 preview 1. The option is off by default even when the code is compiled using the `/permissive-` option (or an option that implies `/permissive-`, such as `/std:c++20` or `/std:c++latest`). To enable the error check, you must explicitly add `/Zc:templateScope` to the compiler command line. To explicitly disable the check, use the `/Zc:templateScope-` option.

Example

Under `/Zc:templateScope`, this sample code produces an error:

C++

```
template<typename T>
void f(T&& t) {
    int T = 13;
}

/* Output:
t.cpp(3): error C7527: 'T': a template parameter name cannot be reused
within its scope
*/
```

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.

3. In **Additional options**, add `/Zc:templateScope` or `/Zc:templateScope-`. Choose **OK** or **Apply** to save your changes.

See also

/Zc (Conformance)
/permissive-
/std (Specify language standard version)

`/Zc:ternary` (Enforce conditional operator rules)

Article • 08/17/2021

Enable enforcement of C++ Standard rules for the types and const or volatile (cv) qualification of the second and third operands in a conditional operator expression.

Syntax

> `/Zc:ternary`[`-`]

Remarks

Starting in Visual Studio 2017, the compiler supports C++ standard *conditional operator* (`?:`) behavior. It's also known as the *ternary operator*. The C++ Standard requires ternary operands satisfy one of three conditions: The operands must be of the same type and `const` or `volatile` qualification (cv-qualification), or only one operand must be unambiguously convertible to the same type and cv-qualification as the other. Or, one or both operands must be a throw expression. In versions before Visual Studio 2017 version 15.5, the compiler allowed conversions that are considered ambiguous by the standard.

When the `/Zc:ternary` option is specified, the compiler conforms to the standard. It rejects code that doesn't satisfy the rules for matched types and cv-qualification of the second and third operands.

The `/Zc:ternary` option is off by default in Visual Studio 2017. Use `/Zc:ternary` to enable conforming behavior, or `/Zc:ternary-` to explicitly specify the previous non-conforming compiler behavior. The /permissive- option implicitly enables this option, but it can be overridden by using `/Zc:ternary-`.

Examples

This sample shows how a class that provides both non-explicit initialization from a type, and conversion to a type, can lead to ambiguous conversions. This code is accepted by the compiler by default, but rejected when `/Zc:ternary` or `/permissive-` is specified.

```
C++
```

```
// zcternary1.cpp
// Compile by using: cl /EHsc /W4 /nologo /Zc:ternary zcternary1.cpp

struct A
{
   long l;
   A(int i) : l{i} {}    // explicit prevents conversion of int
   operator int() const { return static_cast<int>(l); }
};

int main()
{
   A a(42);
   // Accepted when /Zc:ternary (or /permissive-) is not used
   auto x = true ? 7 : a;  // old behavior prefers A(7) over (int)a
   auto y = true ? A(7) : a;   // always accepted
   auto z = true ? 7 : (int)a; // always accepted
   return x + y + z;
}
```

To fix this code, make an explicit cast to the preferred common type, or prevent one direction of type conversion. You can keep the compiler from matching a type conversion by making the conversion explicit.

An important exception to this common pattern is when the type of the operands is one of the null-terminated string types, such as `const char*`, `const char16_t*`, and so on. You can also reproduce the effect with array types and the pointer types they decay to. The behavior when the actual second or third operand to `?:` is a string literal of corresponding type depends on the language standard used. C++17 has changed semantics for this case from C++14. As a result, the compiler accepts the code in the following example under the default **`/std:c++14`**, but rejects it when you specify **`/std:c++17`** or later.

C++

```
// zcternary2.cpp
// Compile by using: cl /EHsc /W4 /nologo /Zc:ternary /std:c++17
zcternary2.cpp

struct MyString
{
   const char * p;
   MyString(const char* s = "") noexcept : p{s} {} // from char*
   operator const char*() const noexcept { return p; } // to char*
};

int main()
{
   MyString s;
```

```
   auto x = true ? "A" : s; // MyString: permissive prefers MyString("A")
over (const char*)s
}
```

To fix this code, cast one of the operands explicitly.

Under `/Zc:ternary`, the compiler rejects conditional operators where one of the arguments is of type `void`, and the other isn't a `throw` expression. A common use of this pattern is in ASSERT-like macros:

C++

```
// zcternary3.cpp
// Compile by using: cl /EHsc /W4 /nologo /Zc:ternary /c zcternary3.cpp

void myassert(const char* text, const char* file, int line);
#define ASSERT(ex) (void)((ex) ? 0 : myassert(#ex, __FILE__, __LINE__))
// To fix, define it this way instead:
// #define ASSERT(ex) (void)((ex) ? void() : myassert(#ex, __FILE__,
__LINE__))

int main()
{
   ASSERT(false);  // C3447
}
```

The typical solution is to replace the non-void argument with `void()`.

This sample shows code that generates an error under both `/Zc:ternary` and `/Zc:ternary-`:

C++

```
// zcternary4.cpp
// Compile by using:
//   cl /EHsc /W4 /nologo /Zc:ternary zcternary4.cpp
//   cl /EHsc /W4 /nologo /Zc:ternary zcternary4.cpp

int main() {
   auto p1 = [](int a, int b) { return a > b; };
   auto p2 = [](int a, int b) { return a > b; };
   auto p3 = true ? p1 : p2; // C2593 under /Zc:ternary, was C2446
}
```

This code previously gave this error:

Output

```
error C2446: ':': no conversion from 'foo::
<lambda_f6cd18702c42f6cd636bfee362b37033>' to 'foo::
<lambda_717fca3fc65510deea10bc47e2b06be4>'
note: No user-defined-conversion operator available that can perform this
conversion, or the operator cannot be called
```

With `/Zc:ternary`, the reason for failure becomes clearer. Any of several implementation-defined calling conventions could be used to generate each lambda. However, the compiler has no preference rule to disambiguate the possible lambda signatures. The new output looks like this:

Output

```
error C2593: 'operator ?' is ambiguous
note: could be 'built-in C++ operator?(bool (__cdecl *)(int,int), bool
(__cdecl *)(int,int))'
note: or       'built-in C++ operator?(bool (__stdcall *)(int,int), bool
(__stdcall *)(int,int))'
note: or       'built-in C++ operator?(bool (__fastcall *)(int,int), bool
(__fastcall *)(int,int))'
note: or       'built-in C++ operator?(bool (__vectorcall *)(int,int), bool
(__vectorcall *)(int,int))'
note: while trying to match the argument list '(foo::
<lambda_717fca3fc65510deea10bc47e2b06be4>, foo::
<lambda_f6cd18702c42f6cd636bfee362b37033>)'
```

A common source of problems found by `/Zc:ternary` comes from conditional operators used in template meta-programming. Some of the result types change under this switch. The following example demonstrates two cases where `/Zc:ternary` changes a conditional expression's result type in a non-meta-programming context:

C++

```
// zcternary5.cpp
// Compile by using: cl /EHsc /W4 /nologo /Zc:ternary zcternary5.cpp

int main(int argc, char**) {
   char a = 'A';
   const char b = 'B';
   decltype(auto) x = true ? a : b; // char without, const char& with
/Zc:ternary
   const char(&z)[2] = argc > 3 ? "A" : "B"; // const char* without
/Zc:ternary
   return x > *z;
}
```

The typical fix is to apply a `std::remove_reference` trait on the result type, where needed to preserve the old behavior.

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/Zc:ternary` or `/Zc:ternary-` and then choose **OK**.

See also

/Zc (Conformance)

/Zc:threadSafeInit (Thread-safe Local Static Initialization)

Article • 08/03/2021

The **/Zc:threadSafeInit** compiler option tells the compiler to initialize static local (function scope) variables in a thread-safe way, eliminating the need for manual synchronization. Only initialization is thread-safe. Use and modification of static local variables by multiple threads must still be manually synchronized. This option is available starting in Visual Studio 2015. By default, Visual Studio enables this option.

Syntax

> **/Zc:threadSafeInit**[-]

Remarks

In the C++11 standard, block scope variables with static or thread storage duration must be zero-initialized before any other initialization takes place. Initialization occurs when control first passes through the declaration of the variable. If an exception is thrown during initialization, the variable is considered uninitialized, and initialization is re-attempted the next time control passes through the declaration. If control enters the declaration concurrently with initialization, the concurrent execution blocks while initialization is completed. The behavior is undefined if control re-enters the declaration recursively during initialization. By default, Visual Studio starting in Visual Studio 2015 implements this standard behavior. This behavior may be explicitly specified by setting the **/Zc:threadSafeInit** compiler option.

The **/Zc:threadSafeInit** compiler option is on by default. The /permissive- option does not affect **/Zc:threadSafeInit**.

Thread-safe initialization of static local variables relies on code implemented in the Universal C run-time library (UCRT). To avoid taking a dependency on the UCRT, or to preserve the non-thread-safe initialization behavior of versions of Visual Studio prior to Visual Studio 2015, use the **/Zc:threadSafeInit-** option. If you know that thread-safety is not required, use this option to generate slightly smaller, faster code around static local declarations.

Thread-safe static local variables use thread-local storage (TLS) internally to provide efficient execution when the static has already been initialized. The implementation of

this feature relies on Windows operating system support functions in Windows Vista and later operating systems. Windows XP, Windows Server 2003, and older operating systems do not have this support, so they do not get the efficiency advantage. These operating systems also have a lower limit on the number of TLS sections that can be loaded. Exceeding the TLS section limit can cause a crash. If this is a problem in your code, especially in code that must run on older operating systems, use **/Zc:threadSafeInit-** to disable the thread-safe initialization code.

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. From the **Configurations** drop down menu, choose **All Configurations**.
3. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
4. Modify the **Additional Options** property to include **/Zc:threadSafeInit** or **/Zc:threadSafeInit-** and then choose **OK**.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
/Zc (Conformance)

/Zc:throwingNew (Assume operator new throws)

Article • 08/03/2021

When the **/Zc:throwingNew** option is specified, the compiler optimizes calls to `operator new` to skip checks for a null pointer return. This option tells the compiler to assume that all linked implementations of `operator new` and custom allocators conform to the C++ standard and throw on allocation failure. By default in Visual Studio, the compiler pessimistically generates null checks (**/Zc:throwingNew-**) for these calls, because users can link with a non-throwing implementation of `operator new` or write custom allocator routines that return null pointers.

Syntax

> **/Zc:throwingNew**[-]

Remarks

Since ISO C++98, the standard has specified that the default operator new throws `std::bad_alloc` when memory allocation fails. Versions of Visual C++ up to Visual Studio 6.0 returned a null pointer on an allocation failure. Beginning in Visual Studio 2002, `operator new` conforms to the standard and throws on failure. To support code that uses the older allocation style, Visual Studio provides a linkable implementation of `operator new` in nothrownew.obj that returns a null pointer on failure. By default, the compiler also generates defensive null checks to prevent these older-style allocators from causing an immediate crash on failure. The **/Zc:throwingNew** option tells the compiler to leave out these null checks, on the assumption that all linked memory allocators conform to the standard. This does not apply to explicit non-throwing `operator new` overloads, which are declared by using an additional parameter of type `std::nothrow_t` and have an explicit `noexcept` specification.

Conceptually, to create an object on the free store, the compiler generates code to allocate its memory and then to invoke its constructor to initialize the memory. Because the MSVC compiler normally cannot tell if this code will be linked to a non-conforming, non-throwing allocator, by default it also generates a null check before calling the constructor. This prevents a null pointer dereference in the constructor call if a non-throwing allocation fails. In most cases, these checks are unnecessary, because the default `operator new` allocators throw instead of returning null pointers. The checks also

have unfortunate side effects. They bloat the code size, they flood the branch predictor, and they inhibit other useful compiler optimizations such as devirtualization or const propagation out of the initialized object. The checks exist only to support code that links to *nothrownew.obj* or has custom non-conforming `operator new` implementations. If you do not use non-conforming `operator new`, we recommend you use **/Zc:throwingNew** to optimize your code.

The **/Zc:throwingNew** option is off by default, and is not affected by the /permissive- option.

If you compile by using link-time code generation (LTCG), you do not need to specify **/Zc:throwingNew**. When your code is compiled by using LTCG, the compiler can detect if the default, conforming `operator new` implementation is used. If so, the compiler leaves out the null checks automatically. The linker looks for the **/ThrowingNew** flag to tell if the implementation of `operator new` is conforming. You can specify this flag to the linker by including this directive in the source for your custom operator new implementation:

```
C++

#pragma comment(linker, "/ThrowingNew")
```

For more information about conformance issues in Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. From the **Configuration** drop down menu, choose **All Configurations**.
3. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
4. Modify the **Additional Options** property to include **/Zc:throwingNew** or **/Zc:throwingNew-** and then choose **OK**.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax
/Zc (Conformance)
noexcept (C++)
Exception Specifications (throw) (C++)
terminate (exception)

`/Zc:tlsGuards` (Check TLS initialization)

Article • 12/12/2022

The `/Zc:tlsGuards` compiler option generates runtime checks for thread local storage (TLS) initialization in DLLs.

Syntax

> `/Zc:tlsGuards`[`-`]

Remarks

The `/Zc:tlsGuards` compiler option enables checks for initialization of thread-local variables in DLLs. Previously, thread-local variables in DLLs weren't correctly initialized. Other than on the thread that loaded the DLL, they weren't initialized before first use on threads that existed before the DLL was loaded. The `/Zc:tlsGuards` option enables code that corrects this defect. Thread-local variables in such a DLL get initialized immediately before their first use on such threads.

The `/Zc:tlsGuards` option is new in Visual Studio 2019 version 16.5. This option is on by default in all compiler modes. The new behavior of testing for initialization on uses of thread-local variables may be disabled by using the `/Zc:tlsGuards-` compiler option. To disable checks for specific thread-local variables, use the [[msvc::no_tls_guard]] attribute.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In **Additional options**, add `/Zc:tlsGuards`. Choose **OK** or **Apply** to save your changes.

See also

/Zc (Conformance)\

`/Zc:trigraphs` (Trigraphs Substitution)

Article • 08/03/2021

When `/Zc:trigraphs` is specified, the compiler replaces a trigraph character sequence by using a corresponding punctuation character.

Syntax

> `/Zc:trigraphs`[`-`]

Remarks

A *trigraph* consists of two consecutive question marks (`??`) followed by a unique third character. The C language standard supports trigraphs for source files that use a character set that doesn't contain convenient graphic representations for some punctuation characters. For example, when trigraphs are enabled, the compiler replaces the `??=` trigraph by using the `#` character. Through C++14, trigraphs are supported as in C. The C++17 standard removes trigraphs from the C++ language. In C++ code, the `/Zc:trigraphs` compiler option enables substitution of trigraph sequences by the corresponding punctuation character. `/Zc:trigraphs-` disables trigraph substitution.

The `/Zc:trigraphs` option is off by default, and the option isn't affected when the /permissive- option is specified.

For a list of C/C++ trigraphs, and an example that shows how to use trigraphs, see Trigraphs.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/Zc:trigraphs` or `/Zc:trigraphs-` and then choose **OK**.

See also

/Zc (Conformance)
Trigraphs

`/Zc:twoPhase-` (disable two-phase name lookup)

Article • 09/28/2022

The `/Zc:twoPhase-` option, under `/permissive-`, tells the compiler to use the original, non-conforming Microsoft C++ compiler behavior to parse and instantiate class templates and function templates.

Syntax

> `/Zc:twoPhase-`

Remarks

Visual Studio 2017 version 15.3 and later: Under /permissive-, the compiler uses two-phase name lookup for template name resolution. If you also specify `/Zc:twoPhase-`, the compiler reverts to its previous non-conforming class template and function template name resolution and substitution behavior. When `/permissive-` isn't specified, the non-conforming behavior is the default.

The Windows SDK header files in version 10.0.15063.0 (Creators Update or RS2) and earlier don't work in conformance mode. `/Zc:twoPhase-` is required to compile code for those SDK versions when you use `/permissive-`. Versions of the Windows SDK starting with version 10.0.15254.0 (Fall Creators Update or RS3) work correctly in conformance mode. They don't require the `/Zc:twoPhase-` option.

Use `/Zc:twoPhase-` if your code requires the old behavior to compile correctly. Strongly consider updating your code to conform to the standard.

Compiler behavior under `/Zc:twoPhase-`

By default, or in Visual Studio 2017 version 15.3 and later when you specify both `/permissive-` and `/Zc:twoPhase-`, the compiler uses this behavior:

- It parses only the template declaration, the class head, and the base class list. The template body is captured as a token stream. No function bodies, initializers, default arguments, or noexcept arguments are parsed. The class template is

pseudo-instantiated on a tentative type to validate that the declarations in the class template are correct. Consider this class template:

```
C++

template <typename T> class Derived : public Base<T> { ... }
```

The template declaration, `template <typename T>`, the class head `class Derived`, and the base-class list `public Base<T>` are parsed, but the template body is captured as a token stream.

- When it parses a function template, the compiler parses only the function signature. The function body is never parsed. Instead, it's captured as a token stream.

As a result, if the template body has syntax errors, but the template never gets instantiated, the compiler doesn't diagnose the errors.

Another effect of this behavior is in overload resolution. Non-standard behavior occurs because of the way the token stream is expanded at the site of instantiation. Symbols that weren't visible at the template declaration may be visible at the point of instantiation. That means they can participate in overload resolution. You may find templates change behavior based on code that wasn't visible at template definition, contrary to the standard.

For example, consider this code:

```
C++

// zctwophase.cpp
// To test options, compile by using
// cl /EHsc /nologo /W4 zctwophase.cpp
// cl /EHsc /nologo /W4 /permissive- zctwophase.cpp
// cl /EHsc /nologo /W4 /permissive- /Zc:twoPhase- zctwophase.cpp

#include <cstdio>

void func(long) { std::puts("Standard two-phase") ;}

template<typename T> void g(T x)
{
    func(0);
}

void func(int) { std::puts("Microsoft one-phase"); }

int main()
{
```

```
    g(6174);
}
```

Here's the output when you use the default mode, conformance mode, and conformance mode with `/Zc:twoPhase-` compiler options:

Windows Command Prompt

```
C:\Temp>cl /EHsc /nologo /W4 zctwophase.cpp && zctwophase
zctwophase.cpp
Microsoft one-phase

C:\Temp>cl /EHsc /nologo /W4 /permissive- zctwophase.cpp && zctwophase
zctwophase.cpp
Standard two-phase

C:\Temp>cl /EHsc /nologo /W4 /permissive- /Zc:twoPhase- zctwophase.cpp &&
zctwophase
zctwophase.cpp
Microsoft one-phase
```

When compiled in conformance mode under `/permissive-`, this program prints "`Standard two-phase`", because the second overload of `func` isn't visible when the compiler reaches the template. If you add `/Zc:twoPhase-`, the program prints "`Microsoft one-phase`". The output is the same as when you don't specify `/permissive-`.

Dependent names are names that depend on a template parameter. These names have lookup behavior that is also different under `/Zc:twoPhase-`. In conformance mode, dependent names aren't bound at the point of the template's definition. Instead, the compiler looks them up when it instantiates the template. For function calls with a dependent function name, the name gets bound to functions visible at the call site in the template definition. Other overloads from argument-dependent lookup are added, both at the point of the template definition, and at the point of template instantiation.

Two-phase lookup consists of two parts: The lookup for non-dependent names during template definition, and the lookup for dependent names during template instantiation. Under `/Zc:twoPhase-`, the compiler doesn't do argument-dependent lookup separately from unqualified lookup. That is, it doesn't do two-phase lookup, so the results of overload resolution may be different.

Here's another example:

C++

```
// zctwophase1.cpp
// To test options, compile by using
// cl /EHsc /W4 zctwophase1.cpp
// cl /EHsc /W4 /permissive- zctwophase1.cpp
// cl /EHsc /W4 /permissive- /Zc:twoPhase- zctwophase1.cpp

#include <cstdio>

void func(long) { std::puts("func(long)"); }

template <typename T> void tfunc(T t) {
    func(t);
}

void func(int) { std::puts("func(int)"); }

namespace NS {
    struct S {};
    void func(S) { std::puts("NS::func(NS::S)"); }
}

int main() {
    tfunc(1729);
    NS::S s;
    tfunc(s);
}
```

When compiled without `/permissive-`, this code prints:

Output

```
func(int)
NS::func(NS::S)
```

When compiled with `/permissive-`, but without `/Zc:twoPhase-`, this code prints:

Output

```
func(long)
NS::func(NS::S)
```

When compiled with both `/permissive-` and `/Zc:twoPhase-`, this code prints:

Output

```
func(int)
NS::func(NS::S)
```

In conformance mode under `/permissive-`, the call `tfunc(1729)` resolves to the `void func(long)` overload. It doesn't resolve to the `void func(int)` overload, as under `/Zc:twoPhase-`. The reason is, the unqualified `func(int)` is declared after the definition of the template, and it isn't found through argument-dependent lookup. But `void func(S)` does participate in argument-dependent lookup, so it's added to the overload set for the call `tfunc(s)`, even though it's declared after the function template.

Update your code for two-phase conformance

Older versions of the compiler don't require the keywords `template` and `typename` everywhere the C++ Standard requires them. These keywords are needed in some positions to disambiguate how compilers should parse a dependent name during the first phase of lookup. For example:

```
T::Foo<a || b>(c);
```

A conforming compiler parses `Foo` as a variable in the scope of `T`, meaning this code is a logical-or expression with `T::foo < a` as the left operand and `b > (c)` as the right operand. If you mean to use `Foo` as a function template, you must indicate that it's a template by adding the `template` keyword:

```
T::template Foo<a || b>(c);
```

In versions Visual Studio 2017 version 15.3 and later, when `/permissive-` and `/Zc:twoPhase-` are specified, the compiler allows this code without the `template` keyword. It interprets the code as a call to a function template with an argument of `a || b`, because it only parses templates in a limited fashion. The code above isn't parsed at all in the first phase. During the second phase, there's enough context to tell that `T::Foo` is a template rather than a variable, so the compiler doesn't enforce use of the keyword.

This behavior can also be seen by eliminating the keyword `typename` before names in function template bodies, initializers, default arguments, and noexcept arguments. For example:

C++

```
template<typename T>
typename T::TYPE func(typename T::TYPE*)
{
    /* typename */ T::TYPE i;
}
```

If you don't use the keyword `typename` in the function body, this code compiles under `/permissive- /Zc:twoPhase-`, but not under `/permissive-` alone. The `typename` keyword is required to indicate that the `TYPE` is dependent. Because the body isn't parsed under `/Zc:twoPhase-`, the compiler doesn't require the keyword. In `/permissive-` conformance mode, code without the `typename` keyword generates errors. To migrate your code to conformance in Visual Studio 2017 version 15.3 and beyond, insert the `typename` keyword where it's missing.

Similarly, consider this code sample:

```
C++
```

```
template<typename T>
typename T::template X<T>::TYPE func(typename T::TYPE)
{
    typename T::/* template */ X<T>::TYPE i;
}
```

Under `/permissive- /Zc:twoPhase-` and in older compilers, the compiler only requires the `template` keyword on line 2. In conformance mode, the compiler now also requires the `template` keyword on line 4 to indicate that `T::X<T>` is a template. Look for code that is missing this keyword, and supply it to make your code conform to the standard.

For more information about conformance issues, see C++ conformance improvements in Visual Studio and Nonstandard behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/Zc:twoPhase-` and then choose **OK**.

See also

/Zc (Conformance)

/Zc:wchar_t (wchar_t Is Native Type)

Article • 08/03/2021

Parse `wchar_t` as a built-in type according to the C++ standard.

Syntax

> **/Zc:wchar_t**[-]

Remarks

If **/Zc:wchar_t** is on, `wchar_t` is a keyword for a built-in integral type in code compiled as C++. If **/Zc:wchar_t-** (with a minus sign) is specified, or in code compiled as C, `wchar_t` is not a built-in type. Instead, `wchar_t` is defined as a `typedef` for `unsigned short` in the canonical header stddef.h. (The Microsoft implementation defines it in another header that is included by stddef.h and other standard headers.)

We do not recommend **/Zc:wchar_t-** because the C++ standard requires that `wchar_t` be a built-in type. Using the `typedef` version can cause portability problems. If you upgrade from earlier versions of Visual Studio and encounter compiler error C2664 because the code is trying to implicitly convert a `wchar_t` to `unsigned short`, we recommend that you change the code to fix the error, instead of setting **/Zc:wchar_t-**.

The **/Zc:wchar_t** option is on by default in C++ compilations, and is ignored in C compilations. The /permissive- option does not affect **/Zc:wchar_t**.

Microsoft implements `wchar_t` as a two-byte unsigned value. It maps to the Microsoft-specific native type `__wchar_t`. For more information about `wchar_t`, see Data Type Ranges and Fundamental Types.

If you write new code that has to interoperate with older code that still uses the `typedef` version of `wchar_t`, you can provide overloads for both the `unsigned short` and `__wchar_t` variations of `wchar_t`, so that your code can be linked with code compiled with **/Zc:wchar_t** or code compiled without it. Otherwise, you would have to provide two different builds of the library, one with and one without **/Zc:wchar_t** enabled. Even in this case, we recommend that you build the older code by using the same compiler that you use to compile the new code. Never mix binaries compiled with different compilers.

When **/Zc:wchar_t** is specified, **_WCHAR_T_DEFINED** and **_NATIVE_WCHAR_T_DEFINED** symbols are defined. For more information, see Predefined Macros.

If your code uses the compiler COM global functions, because **/Zc:wchar_t** is now on by default, we recommend that you change explicit references to comsupp.lib (either from the comment pragma or on the command line) to either comsuppw.lib or comsuppwd.lib. (If you must compile with **/Zc:wchar_t-**, use comsupp.lib.) If you include the comdef.h header file, the correct library is specified for you. For information about compiler COM support, see Compiler COM Support.

The `wchar_t` built-in type is not supported when you compile C code. For more information about conformance issues with Visual C++, see Nonstandard Behavior.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Language** page.
3. Modify the **Treat wchar_t as Built-in Type** property.

To set this compiler option programmatically

- See TreatWChar_tAsBuiltInType.

See also

/Zc (Conformance)

`/Zc:zeroSizeArrayNew` (Call member new/delete on arrays)

Article • 11/09/2022

The `/Zc:zeroSizeArrayNew` compiler option calls member `new` and `delete` for zero-length arrays of objects.

Syntax

> `/Zc:zeroSizeArrayNew`[`-`]

Remarks

The `/Zc:zeroSizeArrayNew` compiler option enables calls to member `new` and `delete` for zero-length arrays of objects of class types with virtual destructors. This behavior conforms to the standard. This compiler option is new in Visual Studio 2019 version 16.9 and is enabled by default in all compiler modes. Previously, in code compiled by versions before Visual Studio 2019 version 16.9, the compiler invoked global `new` and `delete` on zero-length arrays of objects of class types with virtual destructors.

The `/Zc:zeroSizeArrayNew` option may cause a breaking change in code that relied on the previous non-conforming behavior. To restore the previous behavior, use the `/Zc:zeroSizeArrayNew-` compiler option.

To set this compiler option in Visual Studio

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. In **Additional options**, add `/Zc:zeroSizeArrayNew` or `/Zc:zeroSizeArrayNew-`. Choose **OK** or **Apply** to save your changes.

See also

/Zc (Conformance)\

/Zf (Faster PDB generation)

Article • 08/17/2021

Enable faster PDB generation in parallel builds by minimizing RPC calls to mspdbsrv.exe.

Syntax

> **/Zf**

Remarks

The **/Zf** option enables compiler support for faster generation of PDB files when using the /MP (Build with Multiple Processes) option, or when the build system (for example, MSBuild or CMake) may run multiple cl.exe compiler processes at the same time. This option causes the compiler front end to delay generation of type indexes for each type record in the PDB file until the end of compilation, then requests them all in a single RPC call to mspdbsrv.exe, instead of making an RPC request for each record. This can substantially improve build throughput by reducing the RPC load on the mspdbsrv.exe process in an environment where multiple cl.exe compiler processes run simultaneously.

Because the **/Zf** option only applies to PDB generation, it requires the /Zi or /ZI option.

The **/Zf** option is available beginning in Visual Studio 2017 version 15.1, where it is off by default. Starting in Visual Studio 2017 version 15.7, this option is on by default when the **/Zi** or **/ZI** option is enabled.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include **/Zf** and then choose **OK**.

See also

Compiler Options Listed Alphabetically
/MP (Build with Multiple Processes)

/Zg (Generate Function Prototypes)

Article • 08/03/2021

Removed. Creates a function prototype for each function defined in the source file, but does not compile the source file.

Syntax

```
/Zg
```

Remarks

This compiler option is no longer available. It was removed in Visual Studio 2015. This page remains for users of older versions of Visual Studio.

The function prototype includes the function return type and an argument type list. The argument type list is created from the types of the formal parameters of the function. Any function prototypes already present in the source file are ignored.

The list of prototypes is written to standard output. You may find this list helpful to verify that actual arguments and formal parameters of a function are compatible. You can save the list by redirecting standard output to a file. Then you can use **#include** to make the list of function prototypes a part of your source file. Doing so causes the compiler to perform argument type checking.

If you use the **/Zg** option and your program contains formal parameters that have struct, enum, or union type (or pointers to such types), the declaration of each struct, enum, or union type must have a tag (name). In the following sample, the tag name is `MyStruct`.

C

```
// Zg_compiler_option.c
// compile with: /Zg
typedef struct MyStruct { int i; } T2;
void f2(T2 * t) {}
```

The **/Zg** option was deprecated in Visual Studio 2005 and has been removed in Visual Studio 2015. The MSVC compiler has removed support for older, C-style code. For a list

of deprecated compiler options, see **Deprecated and Removed Compiler Options** in Compiler Options Listed by Category.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/ZH` (Hash algorithm for calculation of file checksum in debug info)

Article • 02/01/2022

Specifies which cryptographic hash algorithm to use to generate a checksum of each source file.

Syntax

> `/ZH:MD5`
>
> `/ZH:SHA1`
>
> `/ZH:SHA_256`

Arguments

`/ZH:MD5`
Use an MD5 hash for the checksum. This option is the default in Visual Studio 2019.

`/ZH:SHA1`
Use an SHA-1 hash for the checksum.

`/ZH:SHA_256`
Use an SHA-256 hash for the checksum. This option is the default in Visual Studio 2022 version 17.0 and later.

Remarks

PDB files store a checksum for each source file, compiled into the object code in the associated executable. The checksum allows the debugger to verify that the source code it loads matches the executable. The compiler and debugger support MD5, SHA-1, and SHA-256 hash algorithms. By default, in Visual Studio 2019 the compiler uses an MD5 hash to generate the checksum. To specify this hash algorithm explicitly, use the `/ZH:MD5` option.

Because of a risk of collision problems in MD5 and SHA-1, Microsoft recommends you use the `/ZH:SHA_256` option. The SHA-256 hash might result in a small increase in compile times. The `/ZH:SHA_256` option is the default in Visual Studio 2022 version 17.0 and later.

When more than one `/ZH` option is specified, the last option is used.

The `/ZH` option is available in Visual Studio 2019 version 16.4 and later.

To set this compiler option in the Visual Studio development environment

1. Open the **Property Pages** dialog box for the project. For details, see Set C++ compiler and build properties in Visual Studio.
2. Set the **Configuration** drop-down to **All Configurations**. Set the **Platform** drop-down to **All Platforms**.
3. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
4. Modify the **Additional options** property to add a `/ZH:MD5`, `/ZH:SHA1`, or `/ZH:SHA_256` option, and then choose **OK**.

See also

Compiler options
Source server

/Zl (Omit Default Library Name)

Article • 08/03/2021

Omits the default C runtime library name from the .obj file. By default, the compiler puts the name of the library into the .obj file to direct the linker to the correct library.

Syntax

```
/Zl
```

Remarks

For more information on the default library, see Use Run-Time Library.

You can use **/Zl** to compile .obj files you plan to put into a library. Although omitting the library name saves only a small amount of space for a single .obj file, the total space saved is significant in a library that contains many object modules.

This option is an advanced option. Setting this option removes certain C Runtime library support that may be required by your application, resulting in link-time errors if your application depends on this support. If you use this option you must provide the required components in some other way.

Use /NODEFAULTLIB (Ignore Libraries). to direct the linker to ignore library references in all .obj files.

For more information, see CRT Library Features.

When compiling with **/Zl**, `_VC_NODEFAULTLIB` is defined. For example:

C++

```
// vc_nodefaultlib.cpp
// compile with: /Zl
void Test() {
   #ifdef _VC_NODEFAULTLIB
      int i;
   #endif

   int i;   // C2086
}
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Advanced** property page.
3. Modify the **Omit Default Library Names** property.

To set this compiler option programmatically

- See OmitDefaultLibName.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/Zm` (Specify precompiled header memory allocation limit)

Article • 02/23/2022

Determines the amount of memory that the compiler allocates to construct precompiled headers.

Syntax

> **`/Zm`***`factor`*

Arguments

`factor`
A scaling factor percentage that determines the amount of memory that the compiler uses to construct precompiled headers.

The ***`factor`*** argument is a percentage of the default size of a compiler-defined work buffer. The default value of ***`factor`*** is 100 (percent), but you can specify larger or smaller amounts.

Remarks

In versions before Visual Studio 2015, the C++ compiler used several discrete heaps, and each had a finite limit. Currently, the compiler dynamically grows the heaps as necessary up to a total heap size limit, and allows the precompiled header to comprise multiple address ranges. Now, the **`/Zm`** compiler option is rarely necessary.

If the compiler runs out of heap space and emits the C1060 error message when you use the **`/Zm`** compiler option, you might have reserved too much memory. Consider removing the **`/Zm`** option.

If the compiler emits the C1076 error message, an accompanying C3859 message specifies the ***`factor`*** argument to use when you recompile by using the **`/Zm`** compiler option. This message is only significant when a precompiled header uses `#pragma hdrstop`. In other cases, it's a spurious error caused by Windows virtual memory pressure issues, and the recommendation to use the **`/Zm`** option should be ignored. Instead, consider reducing the number of parallel processes when using the **`/maxcpucount`** option

to MSBUILD.EXE together with the `/MP` option to CL.EXE. For more information, see Precompiled Header (PCH) issues and recommendations.

The following table shows how the *`factor`* argument affects the memory allocation limit. In the table, we assume the size of the default precompiled header buffer is 75 MB.

Value of *`factor`*	**Memory allocation limit**
10	7.5 MB
100	75 MB
200	150 MB
1000	750 MB
2000	1500 MB

Other ways to set the memory allocation limit

To set the `/Zm` compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the **/Zm** compiler option in the **Additional Options** box.

To set the `/Zm` compiler option programmatically

- See AdditionalOptions.

See also

MSVC compiler options
MSVC compiler command-line syntax

/Zo (Enhance Optimized Debugging)

Article • 06/01/2022

Generate enhanced debugging information for optimized code in non-debug builds.

Syntax

> `/Zo`[`-`]

Remarks

The `/Zo` compiler option generates enhanced debugging information for optimized code. Optimization may use registers for local variables, reorder code, vectorize loops, and inline function calls. These optimizations can obscure the relationship between the source code and the compiled object code. The `/Zo` option tells the compiler to generate extra debugging information for local variables and inlined functions. It allows you to see variables in the **Autos**, **Locals**, and **Watch** windows when you step through optimized code in the Visual Studio debugger. It also enables stack traces to show inlined functions in the WinDBG debugger. Debug builds that have disabled optimizations (/Od) don't need the extra debugging information generated when `/Zo` is specified. Use the `/Zo` option to debug Release configurations with optimization turned on. For more information on optimization options, see /O options (Optimize Code).

The `/Zo` option is enabled by default when you specify debugging information with `/Zi` or `/Z7`. It's disabled by the `/ZI` compiler option. Specify `/Zo-` to explicitly disable this compiler option.

The `/Zo` option is available starting in Visual Studio 2013 Update 3, and it replaces the previously undocumented `/d2Zi+` option.

To set the `/Zo` compiler option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Modify the **Additional Options** property to include `/Zo` and then choose **OK**.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/O Options (Optimize code)
/Z7, /Zi, /ZI (Debug information format)
Edit and Continue

/Zp (Struct Member Alignment)

Article • 07/06/2022

Controls how the members of a structure are packed into memory and specifies the same packing for all structures in a module.

Syntax

> `/Zp`[`1`|`2`|`4`|`8`|`16`]

Remarks

The `/ZpN` option tells the compiler where to store each structure member. The compiler stores members after the first one on a boundary that's the smaller of either the size of the member type, or an *N*-byte boundary.

The available packing values are described in the following table:

/Zp argument	Effect
1	Packs structures on 1-byte boundaries. Same as `/Zp`.
2	Packs structures on 2-byte boundaries.
4	Packs structures on 4-byte boundaries.
8	Packs structures on 8-byte boundaries (default for x86, ARM, and ARM64).
16	Packs structures on 16-byte boundaries (default for x64 and ARM64EC).

Don't use this option unless you have specific alignment requirements.

> ⚠ **Warning**
>
> The C/C++ headers in the Windows SDK assume the platform's default alignment is used. Don't change the setting from the default when you include the Windows SDK headers, either by using `/Zp` on the command line or by using `#pragma pack`. Otherwise, your application may cause memory corruption at runtime.

You can also use the pack pragma to control structure packing. For more information about alignment, see:

- align
- alignof Operator
- __unaligned
- /ALIGN (Section Alignment)

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Code Generation** property page.
3. Modify the **Struct Member Alignment** property.

To set this compiler option programmatically

- See StructMemberAlignment.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

/Zs (Syntax Check Only)

Article • 08/03/2021

Tells the compiler to check only the syntax of the source files on the command line.

Syntax

```
/Zs
```

Remarks

When using this option, no output files are created, and error messages are written to standard output.

The **/Zs** option provides a quick way to find and correct syntax errors before you compile and link a source file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **C/C++** > **Command Line** property page.
3. Enter the compiler option in the **Additional Options** box.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

`/ZW` (Windows Runtime Compilation)

Article • 06/26/2023

Compiles source code to support Microsoft C++ component extensions C++/CX for the creation of Universal Windows Platform (UWP) apps.

When you use `/ZW` to compile, always specify `/EHsc` as well.
`/ZW` isn't compatible with `/std:c++20`.

Syntax

```
C++

/ZW /EHsc
/ZW:nostdlib /EHsc
```

Arguments

`nostdlib`
Indicates that `Platform.winmd`, `Windows.Foundation.winmd`, and other default Windows metadata (`.winmd`) files aren't automatically included in the compilation. Instead, you must use the /FU (Name Forced #using File) compiler option to explicitly specify Windows metadata files.

Remarks

When you specify the `/ZW` option, the compiler supports these features:

- The required metadata files, namespaces, data types, and functions that your app requires to execute in the Windows Runtime.
- Automatic reference-counting of Windows Runtime objects, and automatic discarding of an object when its reference count goes to zero.

Because the incremental linker doesn't support the Windows metadata included in .obj files by using the `/ZW` option, the deprecated /Gm (Enable Minimal Rebuild) option is incompatible with `/ZW`.

For more information, see Visual C++ Language Reference.

Requirements

See also

MSVC Compiler Options
MSVC Compiler Command-Line Syntax

Structured SARIF Diagnostics

Article • 11/13/2023

The MSVC compiler can be made to output diagnostics as SARIF (Static Analysis Results Interchange Format). SARIF is a machine-readable JSON-based format.

There are two ways to make the MSVC compiler produce SARIF diagnostics:

- Pass the `/experimental:log` switch on the command line. See the documentation for /experimental:log for details.
- Launch `cl.exe` programatically and set the `SARIF_OUTPUT_PIPE` environment variable to retrieve SARIF blocks through a pipe.

Retrieving SARIF through a pipe

Tools that consume SARIF from the MSVC compiler while a compilation is in progress use a pipe. See the documentation for CreatePipe for details about creating Windows pipes.

To retrieve SARIF through a pipe, set the `SARIF_OUTPUT_PIPE` environment variable to be the UTF-16-encoded integer representation of the `HANDLE` to the write end of the pipe, then launch `cl.exe`. SARIF is sent along the pipe as follows:

- When a new diagnostic is available, it is written to this pipe.
- Diagnostics are written to the pipe one-at-a-time rather than as an entire SARIF object.
- Each diagnostic is represented by a JSON-RPC 2.0 message of type Notification.
- The JSON-RPC message is prefixed with a `Content-Length` header with the form `Content-Length: <N>` followed by two newlines, where `<N>` is the length of the following JSON-RPC message in bytes.
- The JSON-RPC message and header are both encoded in UTF-8.
- This JSON-RPC-with-header format is compatible with vs-streamjsonrpc.
- The method name for the JSON-RPC call is `OnSarifResult`.
- The call has a single parameter that is encoded by-name with the parameter name `result`.
- The value of the argument is a single `result` object as specified by the SARIF Version 2.1 standard.

Example

Here's an example of a JSON-RPC SARIF result produced by `cl.exe`:

JSON

```
Content-Length: 334

{"jsonrpc":"2.0","method":"OnSarifResult","params":{"result":
{"ruleId":"C1034","level":"fatal","message":{"text":"iostream: no include
path set"},"locations":[{"physicalLocation":{"artifactLocation":
{"uri":"file:///C:/Users/sybrand/source/repos/cppcon-diag/cppcon-
diag/cppcon-diag.cpp"},"region":{"startLine":1,"startColumn":10}}}]}}}
{"jsonrpc":"2.0","method":"OnSarifResult","params":{"result":
{"ruleId":"C1034","level":"fatal","message":{"text":"iostream: no include
path set"},"locations":[{"physicalLocation":{"artifactLocation":
{"uri":"file:///C:/Users/sybrand/source/repos/cppcon-diag/cppcon-
diag/cppcon-diag.cpp"},"region":{"startLine":1,"startColumn":10}}}]}}}
```

SARIF result data

The compiler outputs SARIF that may include additional information to represent the nested structure of some diagnostics. A diagnostic (represented by a `result` SARIF object) may contain a "diagnostic tree" of additional information in its `relatedLocations` field. This tree is encoded using a SARIF property bag as follows:

A `location` object's `properties` field may contain a `nestingLevel` property whose value is the depth of this location in the diagnostic tree. If a location doesn't have a `nestingLevel` specified, the depth is considered to be `0` and this location is a child of the root diagnostic represented by the `result` object containing it. Otherwise, if the value is greater than the depth of the location immediately preceding this location in the `relatedLocations` field, this location is a child of that location. Otherwise, this location is a sibling of the closest preceding `location` in the `relatedLocations` field with the same depth.

Example

Consider the following code:

C++

```
struct dog {};
struct cat {};

void pet(dog);
void pet(cat);
```

```
struct lizard {};

int main() {
    pet(lizard{});
}
```

When this code is compiled, the compiler produces the following `result` object (`physicalLocation` properties have been removed for brevity):

JSON


```
{
    "ruleId": "C2665",
    "level": "error",
    "message": {
        "text": "'pet': no overloaded function could convert all the
argument types"
    },
    "relatedLocations": [
        {
            "id": 0,
            "message": {
                "text": "could be 'void pet(cat)'"
            }
        },
        {
            "id": 1,
            "message": {
                "text": "'void pet(cat)': cannot convert argument 1 from
'lizard' to 'cat'"
            },
            "properties": {
                "nestingLevel": 1
            }
        },
        {
            "id": 2,
            "message": {
                "text": "No user-defined-conversion operator available that
can perform this conversion, or the operator cannot be called"
            },
            "properties": {
                "nestingLevel": 2
            }
        },
        {
            "id": 3,
            "message": {
                "text": "or       'void pet(dog)'"
            }
        },
        {
            "id": 4,
```

```
            "message": {
                "text": "'void pet(dog)': cannot convert argument 1 from
'lizard' to 'dog'"
            },
            "properties": {
                "nestingLevel": 1
            }
        },
        {
            "id": 5,
            "message": {
                "text": "No user-defined-conversion operator available that
can perform this conversion, or the operator cannot be called"
            },
            "properties": {
                "nestingLevel": 2
            }
        },
        {
            "id": 6,
            "message": {
                "text": "while trying to match the argument list '(lizard)'"
            }
        }
    ]
}
```


The logical diagnostics tree produced from the messages in this `result` object is:

- 'pet': no overloaded function could convert all the argument types
 - could be 'void pet(cat)'
 - 'void pet(cat)': cannot convert argument 1 from 'lizard' to 'cat
 - No user-defined-conversion operator available that can perform this conversion, or the operator cannot be called
 - or 'void pet(dog)'
 - 'void pet(dog)': cannot convert argument 1 from 'lizard' to 'dog'
 - No user-defined-conversion operator available that can perform this conversion, or the operator cannot be called
 - while trying to match the argument list '(lizard)'

See also

/experimental:log (Enable structured SARIF diagnostics)

Unicode support in the compiler and linker

Article • 08/03/2021

Most Microsoft C/C++ (MSVC) build tools support Unicode inputs and outputs.

Filenames

Filenames specified on the command line or in compiler directives (such as `#include`) may contain Unicode characters.

Source code files

Unicode characters are supported in identifiers, macros, string and character literals, and in comments. Universal character names are also supported.

Unicode can be input into a source code file in the following encodings:

- UTF-16 little endian with or without byte order mark (BOM)
- UTF-16 big endian with or without BOM
- UTF-8 with BOM

In the Visual Studio IDE, you can save files in several encoding formats, including Unicode ones. Save them in the **Save File As** dialog by using the dropdown on the **Save** button. Select **Save with Encoding** in the dropdown. Then, in the **Advanced Save Options** dialog, select an encoding from the dropdown list. Choose **OK** to save the file.

Output

During compilation, the compiler outputs diagnostics to the console in UTF-16. The characters that can be displayed at your console depend on the console window properties. Compiler output redirected to a file is in the current ANSI console codepage.

Linker response files and `.DEF` files

Response files and *`.DEF`* files can be either UTF-16 or UTF-8 with a BOM, or ANSI.

`.asm` and `.cod` dumps

`.asm` and *`.cod`* dumps are in ANSI by default for compatibility with MASM. Use /FAu to output UTF-8.

If you specify **`/FAs`**, the intermingled source gets printed directly. It may look garbled, for example, when the source code is UTF-8 and you didn't specify **`/FAsu`**.

See also

Use the MSVC toolset from the command line

Linking

Article • 01/10/2024

In a C++ project, the *linking* step is performed after the compiler compiles the source code into object files (*.obj). The linker (`link.exe`) combines the object files into a single executable file.

Linker options can be set inside or outside of Visual Studio. Within Visual Studio, you access linker options by right-clicking on a project node in **Solution Explorer** and choosing **Properties** to display the property pages. Choose **Linker** in the left pane to expand the node and see all the options.

Linker command-line syntax

When you run the linker outside of Visual Studio, you can specify input in one or more ways:

- On the command line
- Using command files
- In environment variables

The linker first processes options specified in the `LINK` environment variable, followed by options in the order they're specified on the command line and in command files. If an option is repeated with different arguments, the last one processed takes precedence.

Options apply to the entire build; no options can be applied to specific input files.

To run `link.exe`, use the following command syntax:

Windows Command Prompt

```
link arguments
```

The `arguments` include options and filenames and can be specified in any order. Options are processed first, then files. Use one or more spaces or tabs to separate arguments.

> **Note**
>
> You can start this tool only from the Visual Studio command prompt. You can't start it from a system command prompt or from File Explorer.

Command line

On the command line, an option consists of an option specifier, either a dash (`-`) or a forward slash (`/`), followed by the name of the option. Option names can't be abbreviated. Some options take an argument, specified after a colon (`:`). No spaces or tabs are allowed within an option specification, except within a quoted string in the `/COMMENT` option. Specify numeric arguments in decimal or C-language notation. Option names and their keyword or filename arguments aren't case sensitive, but identifiers as arguments are case sensitive.

To pass a file to the linker, specify the filename on the command line after the `link.exe` command. You can specify an absolute or relative path with the filename, and you can use wildcards in the filename. If you omit the dot (`.`) and filename extension, the linker assumes an extension of `.obj` to find the file. The linker doesn't use filename extensions or the lack of them to make assumptions about the contents of files. It determines the type of file by examining it, and processes it accordingly.

The linker returns zero for success (no errors). Otherwise, it returns the error number that stopped the link. For example, if the linker generates `LNK1104`, the linker returns 1104. Accordingly, the lowest error number returned on an error by the linker is 1000. A return value of 128 represents a configuration problem with either the operating system or a .config file; the loader didn't load either `link.exe` or `c2.dll`.

Linker command files

You can pass command-line arguments to `link.exe` in the form of a command file. To specify a command file to the linker, use the following syntax:

```
link @commandfile
```

The *`commandfile`* is the name of a text file. No space or tab is allowed between the at sign (@) and the filename. There's no default extension; you must specify the full filename, including any extension. Wildcards can't be used. You can specify an absolute or relative path with the filename. The linker doesn't use an environment variable to search for the file.

In the command file, arguments are separated by spaces or tabs (as on the command line) and by newline characters.

You can specify all or part of the command line in a command file. You can use more than one command file in a `link.exe` command. The linker accepts the command-file

input as if it was specified in that location on the command line. Command files can't be nested. The linker echoes the contents of command files, unless /NOLOGO is specified.

Example

The following command builds a DLL. It passes the names of object files and libraries in separate command files and uses a third command file for specification of the `/EXPORTS` option:

Windows Command Prompt

```
link /dll @objlist.txt @liblist.txt @exports.txt
```

LINK environment variables

The linker recognizes the following environment variables:

- `LINK` and `_LINK_`, if defined. The linker prepends the options and arguments defined in the `LINK` environment variable and appends the options and arguments defined in the `_LINK_` environment variable to the command line arguments before processing.
- `LIB`, if defined. The linker uses the `LIB` path when it searches for an object, library, or other file specified on the command line or by the /BASE option. It also uses the `LIB` path to find a `.pdb` file named in an object. The `LIB` variable can contain one or more path specifications, separated by semicolons. One path must point to the `\lib` subdirectory of your Visual C++ installation.
- `PATH`, if the tool needs to run `CVTRES` and can't find the file in the same directory as `link.exe` itself. (`link.exe` requires `CVTRES` to link a `.res` file.) `PATH` must point to the `\bin` subdirectory of your Visual C++ installation.
- `TMP`, to specify a directory when linking OMF or `.res` files.

See also

C/C++ Building Reference
MSVC Linker Options
Module-Definition (.def) Files
Linker Support for Delay-Loaded DLLs

Linker options

Article • 06/03/2024

LINK.exe links Common Object File Format (COFF) object files and libraries to create an executable (EXE) file or a dynamic-link library (DLL).

The following table lists options for LINK.exe. For more information about LINK, see:

- Compiler-controlled LINK options
- LINK input files
- LINK output
- Reserved words

On the command line, linker options aren't case-sensitive; for example, `/base` and `/BASE` mean the same thing. For details on how to specify each option on the command line or in Visual Studio, see the documentation for that option.

You can use the comment pragma to specify some linker options.

Linker options listed alphabetically

Expand table

Option	Purpose
@	Specifies a response file.
/ALIGN	Specifies the alignment of each section.
/ALLOWBIND	Specifies that a DLL can't be bound.
/ALLOWISOLATION	Specifies behavior for manifest lookup.
/APPCONTAINER	Specifies whether the app must run within an appcontainer process environment.
/ARM64XFUNCTIONPADMINX64	Specifies the minimum number of bytes of padding between x64 functions in ARM64X images.[17.8]
/ASSEMBLYDEBUG	Adds the DebuggableAttribute to a managed image.
/ASSEMBLYLINKRESOURCE	Creates a link to a managed resource.
/ASSEMBLYMODULE	Specifies that a Microsoft intermediate language (MSIL) module should be imported into the assembly.

Option	Purpose
/ASSEMBLYRESOURCE	Embeds a managed resource file in an assembly.
/BASE	Sets a base address for the program.
/CETCOMPAT	Marks the binary as CET Shadow Stack compatible.
/CGTHREADS	Sets number of cl.exe threads to use for optimization and code generation when link-time code generation is specified.
/CLRIMAGETYPE	Sets the type (IJW, pure, or safe) of a CLR image.
/CLRSUPPORTLASTERROR	Preserves the last error code of functions that are called through the P/Invoke mechanism.
/CLRTHREADATTRIBUTE	Specifies the threading attribute to apply to the entry point of your CLR program.
/CLRUNMANAGEDCODECHECK	Specifies whether the linker applies the `SuppressUnmanagedCodeSecurity` attribute to linker-generated P/Invoke stubs that call from managed code into native DLLs.
/DEBUG	Creates debugging information.
/DEBUGTYPE	Specifies which data to include in debugging information.
/DEF	Passes a module-definition (.def) file to the linker.
/DEFAULTLIB	Searches the specified library when external references are resolved.
/DELAY	Controls the delayed loading of DLLs.
/DELAYLOAD	Causes the delayed loading of the specified DLL.
/DELAYSIGN	Partially signs an assembly.
/DEPENDENTLOADFLAG	Sets default flags on dependent DLL loads.
/DLL	Builds a DLL.
/DRIVER	Creates a kernel mode driver.
/DYNAMICBASE	Specifies whether to generate an executable image that's rebased at load time by using the address space layout randomization (ASLR) feature.
/ENTRY	Sets the starting address.
/ERRORREPORT	Deprecated. Error reporting is controlled by Windows Error Reporting (WER) settings.

Option	Purpose
/EXPORT	Exports a function.
/FILEALIGN	Aligns sections within the output file on multiples of a specified value.
/FIXED	Creates a program that can be loaded only at its preferred base address.
/FORCE	Forces a link to complete even with unresolved symbols or symbols defined more than once.
/FUNCTIONPADMIN	Creates an image that can be hot patched.
/GENPROFILE, /FASTGENPROFILE	Both of these options specify generation of a `.pgd` file by the linker to support profile-guided optimization (PGO). /GENPROFILE and /FASTGENPROFILE use different default parameters.
/GUARD	Enables Control Flow Guard protection.
/HEAP	Sets the size of the heap, in bytes.
/HIGHENTROPYVA	Specifies support for high-entropy 64-bit address space layout randomization (ASLR).
/IDLOUT	Specifies the name of the `.idl` file and other MIDL output files.
/IGNORE	Suppresses output of specified linker warnings.
/IGNOREIDL	Prevents the processing of attribute information into an `.idl` file.
/ILK	Overrides the default incremental database file name.
/IMPLIB	Overrides the default import library name.
/INCLUDE	Forces symbol references.
/INCREMENTAL	Controls incremental linking.
/INFERASANLIBS	Uses inferred sanitizer libraries.
/INTEGRITYCHECK	Specifies that the module requires a signature check at load time.
/KERNEL	Create a kernel mode binary.
/KEYCONTAINER	Specifies a key container to sign an assembly.
/KEYFILE	Specifies a key or key pair to sign an assembly.

Option	Purpose
/LARGEADDRESSAWARE	Tells the compiler that the application supports addresses larger than 2 gigabytes
/LIBPATH	Specifies a path to search before the environmental library path.
/LINKREPRO	Specifies a path to generate link repro artifacts in.
LINKREPROFULLPATHRSP	Generates a response file containing the absolute paths to all the files that the linker took as input.
/LINKREPROTARGET	Generates a link repro only when producing the specified target.[16.1]
/LTCG	Specifies link-time code generation.
/MACHINE	Specifies the target platform.
/MANIFEST	Creates a side-by-side manifest file and optionally embeds it in the binary.
/MANIFESTDEPENDENCY	Specifies a <dependentAssembly> section in the manifest file.
/MANIFESTFILE	Changes the default name of the manifest file.
/MANIFESTINPUT	Specifies a manifest input file for the linker to process and embed in the binary. You can use this option multiple times to specify more than one manifest input file.
/MANIFESTUAC	Specifies whether User Account Control (UAC) information is embedded in the program manifest.
/MAP	Creates a mapfile.
/MAPINFO	Includes the specified information in the mapfile.
/MERGE	Combines sections.
/MIDL	Specifies MIDL command-line options.
/NATVIS	Adds debugger visualizers from a Natvis file to the program database (PDB).
/NOASSEMBLY	Suppresses the creation of a .NET Framework assembly.
/NODEFAULTLIB	Ignores all (or the specified) default libraries when external references are resolved.
/NOENTRY	Creates a resource-only DLL.

Option	Purpose
/NOFUNCTIONPADSECTION	Disables function padding for functions in the specified section.[17.8]
/NOLOGO	Suppresses the startup banner.
/NXCOMPAT	Marks an executable as verified to be compatible with the Windows Data Execution Prevention feature.
/OPT	Controls LINK optimizations.
/ORDER	Places COMDATs into the image in a predetermined order.
/OUT	Specifies the output file name.
/PDB	Creates a PDB file.
/PDBALTPATH	Uses an alternate location to save a PDB file.
/PDBSTRIPPED	Creates a PDB file that has no private symbols.
/PGD	Specifies a `.pgd` file for profile-guided optimizations.
/POGOSAFEMODE	**Obsolete** Creates a thread-safe PGO instrumented build.
/PROFILE	Produces an output file that can be used with the Performance Tools profiler.
/RELEASE	Sets the Checksum in the `.exe` header.
/SAFESEH	Specifies that the image will contain a table of safe exception handlers.
/SECTION	Overrides the attributes of a section.
/SOURCELINK	Specifies a SourceLink file to add to the PDB.
/STACK	Sets the size of the stack in bytes.
/STUB	Attaches an MS-DOS stub program to a Win32 program.
/SUBSYSTEM	Tells the operating system how to run the `.exe` file.
/SWAPRUN	Tells the operating system to copy the linker output to a swap file before it's run.
/TIME	Output linker pass timing information.
/TLBID	Specifies the resource ID of the linker-generated type library.
/TLBOUT	Specifies the name of the `.tlb` file and other MIDL output files.

Option	Purpose
/TSAWARE	Creates an application that is designed specifically to run under Terminal Server.
/USEPROFILE	Uses profile-guided optimization training data to create an optimized image.
/VERBOSE	Prints linker progress messages.
/VERSION	Assigns a version number.
/WHOLEARCHIVE	Includes every object file from specified static libraries.
/WINMD	Enables generation of a Windows Runtime Metadata file.
/WINMDFILE	Specifies the file name for the Windows Runtime Metadata (winmd) output file that's generated by the /WINMD linker option.
/WINMDKEYFILE	Specifies a key or key pair to sign a Windows Runtime Metadata file.
/WINMDKEYCONTAINER	Specifies a key container to sign a Windows Metadata file.
/WINMDDELAYSIGN	Partially signs a Windows Runtime Metadata (`.winmd`) file by placing the public key in the winmd file.
/WX	Treats linker warnings as errors.

[16.1] This option is available starting in Visual Studio 2019 version 16.1.
[17.8] This option is available starting in Visual Studio 2022 version 17.8.

See also

C/C++ building reference
MSVC linker reference

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Compiler-Controlled LINK Options

Article • 08/03/2021

The CL compiler automatically calls LINK unless you specify the /c option. CL provides some control over the linker through command-line options and arguments. The following table summarizes the features in CL that affect linking.

CL specification	**CL action that affects LINK**
Any file name extension other than .c, .cxx, .cpp, or .def	Passes a file name as input to LINK
filename.def	Passes /DEF:*filename*.def
/F*number*	Passes /STACK:*number*
/Fd*filename*	Passes /PDB:*filename*
/Fe*filename*	Passes /OUT:*filename*
/Fm*filename*	Passes /MAP:*filename*
/Gy	Creates packaged functions (COMDATs); enables function-level linking
/LD	Passes /DLL
/LDd	Passes /DLL
/link	Passes remainder of command line to LINK
/MD or /MT	Places a default library name in the .obj file
/MDd or /MTd	Places a default library name in the .obj file. Defines the symbol **_DEBUG**
/nologo	Passes /NOLOGO
/Zd	Passes /DEBUG
/Zi or /Z7	Passes /DEBUG
/Zl	Omits default library name from .obj file

For more information, see MSVC Compiler Options.

See also

MSVC linker reference

MSVC Linker Options

LINK input files

Article • 09/08/2022

You provide the linker with files that contain objects, import and standard libraries, resources, module definitions, and command input. LINK doesn't use file extensions to make assumptions about the contents of a file. Instead, LINK examines each input file to determine what kind of file it is.

Object files on the command line are processed in the order they appear on the command line. Libraries are searched in command line order as well, with the following caveat: Symbols that are unresolved when bringing in an object file from a library are searched for in that library first, and then the following libraries from the command line and /DEFAULTLIB (Specify default library) directives, and then to any libraries at the beginning of the command line.

> **Note**
>
> LINK no longer accepts a semicolon (or any other character) as the start of a comment in response files and order files. Semicolons are recognized only as the start of comments in module-definition files (`.def`).

LINK uses the following types of input files:

- .obj files
- .netmodule files
- .lib files
- .exp files
- .def files
- .pdb files
- .res files
- .exe files
- .txt files
- .ilk files

See also

MSVC linker reference
MSVC linker options

.Obj Files as Linker Input

Article • 08/03/2021

The linker tool (LINK.EXE) accepts .obj files that are in Common Object File Format (COFF).

Remarks

Microsoft provides a complete description of the common object file format. For more information, see PE Format.

Unicode support

Starting with Visual Studio 2005, the Microsoft MSVC compiler supports Unicode characters in identifiers as defined by the ISO/IEC C and C++ standards. Previous versions of the compiler supported only ASCII characters in identifiers. To support Unicode in the names of functions, classes, and statics, the compiler and linker use the Unicode UTF-8 encoding for COFF symbols in .obj files. The UTF-8 encoding is upwardly compatible with the ASCII encoding used by earlier versions of Visual Studio.

For more information about the compiler and linker, see Unicode Support in the Compiler and Linker. For more information about the Unicode standard, see the Unicode organization.

See also

LINK Input Files
MSVC Linker Options
Support for Unicode
Unicode Support in the Compiler and Linker
Unicode standard
PE Format

.netmodule files as linker input

Article • 02/24/2023

link.exe accepts MSIL `.obj` and `.netmodule` files as input. The output file produced by the linker is an assembly or a `.netmodule` file with no run-time dependency on any of the `.obj` or `.netmodule` files that were input to the linker.

Remarks

`.netmodule` files are created by the MSVC compiler with /LN (Create MSIL module) or by the linker with /NOASSEMBLY (Create a MSIL Module). `.obj` files are always created in a C++ compilation. For other Visual Studio compilers, use the **/target:module** compiler option.

The linker must be passed the `.obj` file from the C++ compilation that created the `.netmodule`. Passing in a `.netmodule` is no longer supported because the **/clr:pure** and **/clr:safe** compiler options are deprecated in Visual Studio 2015 and unsupported in Visual Studio 2017 and later.

For information on how to invoke the linker from the command line, see Linker command-line syntax and Use the MSVC toolset from the command line.

Passing a `.netmodule` or `.dll` file to the linker that was compiled by the MSVC compiler with **/clr** can result in a linker error. For more information, see Choosing the format of .netmodule input files.

The linker accepts both native `.obj` files and MSIL `.obj` files compiled with **/clr**. You can pass mixed `.obj` files in the same build. The resulting output file's default verifiability is the same as the lowest input module's verifiability.

You can change an application that's composed of two or more assemblies to be contained in one assembly. Recompile the assemblies' sources, and then link the `.obj` files or `.netmodule` files to produce a single assembly.

Specify an entry point using /ENTRY (Entry-point symbol) when creating an executable image.

When linking with an MSIL `.obj` or `.netmodule` file, use /LTCG (Link-time code generation), otherwise when the linker encounters the MSIL `.obj` or `.netmodule`, it will restart the link with **/LTCG**. You'll see an informational message that the link is restarting. You can ignore this message, but to improve linker performance, explicitly specify **/LTCG**.

MSIL `.obj` or `.netmodule` files can also be passed to cl.exe.

Input MSIL `.obj` or `.netmodule` files can't have embedded resources. Embed resources in an output module or assembly file by using the /ASSEMBLYRESOURCE (Embed a managed resource) linker option. Or, use the **/resource** compiler option in other Visual Studio compilers.

Examples

In C++ code, the `catch` block of a corresponding `try` will be invoked for a non-`System` exception. However, by default, the CLR wraps non-`System` exceptions with RuntimeWrappedException. When an assembly is created from C++ and non-C++ modules, and you want a `catch` block in C++ code to be invoked from its corresponding `try` clause when the `try` block throws a non-`System` exception, you must add the `[assembly:System::Runtime::CompilerServices::RuntimeCompatibility(WrapNonExceptionThrows=false)]` attribute to the source code for the non-C++ modules.

C++

```
// MSIL_linking.cpp
// compile with: /c /clr
value struct V {};

ref struct MCPP {
   static void Test() {
      try {
         throw (gcnew V);
      }
      catch (V ^) {
         System::Console::WriteLine("caught non System exception in C++
source code file");
      }
   }
};

/*
int main() {
   MCPP::Test();
}
*/
```

By changing the `Boolean` value of the `WrapNonExceptionThrows` attribute, you modify the ability of the C++ code to catch a non-`System` exception.

C#

```
// MSIL_linking_2.cs
// compile with: /target:module /addmodule:MSIL_linking.obj
// post-build command: link /LTCG MSIL_linking.obj MSIL_linking_2.netmodule
/entry:MLinkTest.Main /out:MSIL_linking_2.exe /subsystem:console
using System.Runtime.CompilerServices;

// enable non System exceptions
[assembly:RuntimeCompatibility(WrapNonExceptionThrows=false)]

class MLinkTest {
   public static void Main() {
      try {
         MCPP.Test();
      }
      catch (RuntimeWrappedException) {
         System.Console.WriteLine("caught a wrapped exception in C#");
      }
   }
}
```

Output

```
caught non System exception in C++ source code file
```

See also

- LINK Input Files
- MSVC Linker Options

Choosing the Format of .netmodule Input Files

Article • 08/03/2021

An MSIL .obj file (compiled with /clr) can also be used as a .netmodule file. .obj files contain metadata and native symbols. .netmodules only contain metadata.

You can pass an MSIL .obj file to any other Visual Studio compiler via the /addmodule compiler option (but be aware that the .obj file becomes part of the resulting assembly and must be shipped with the assembly). For example, Visual C# and Visual Basic have the /addmodule compiler option.

> **Note**
>
> In most cases, you will need to pass to the linker the .obj file from the compilation that created the .net module. Passing a .dll or .netmodule MSIL module file to the linker may result in LNK1107.

.obj files, along with their associated .h files, which you reference via #include in source, allow C++ applications to consume the native types in the module, whereas in a .netmodule file, only the managed types can be consumed by a C++ application. If you attempt to pass a .obj file to #using, information about native types will not be available; #include the .obj file's .h file instead.

Other Visual Studio compilers can only consume managed types from a module.

Use the following to determine whether you need to use a .netmodule or a .obj file as module input to the MSVC linker:

- If you are building with a Visual Studio compiler other than Visual C++, produce a .netmodule and use the .netmodule as input to the linker.
- If you are using the MSVC compiler to produce modules and if the module(s) will be used to build something other than a library, use the .obj files produced by the compiler as module input to the linker; do not use the .netmodule file as input.
- If your modules will be used to build a native (not a managed) library, use .obj files as module input to the linker and generate a .lib library file.
- If your modules will be used to build a managed library, and if all module input to the linker will be verifiable (produced with /clr:safe), use .obj files as module input

to the linker and generate a .dll (assembly) or .netmodule (module) library file.

- If your modules will be used to build a managed library, and if one or more modules input to the linker will be produced with just /clr, use .obj files as module input to the linker and generate a .dll (assembly). If you want to expose managed types from the library and if you also want C++ applications to consume the native types in the library, your library will consist of the .obj files for the libraries component modules (you will also want to ship the .h files for each module, so they can be referenced with #include from source code).

See also

.netmodule Files as Linker Input

`.lib` files as linker input

Article • 09/22/2022

LINK accepts COFF standard libraries and COFF import libraries, both of which usually have the extension `.lib`. Standard libraries contain objects and are created by the LIB tool. Import libraries contain information about exports in other programs and are created either by LINK when it builds a program that contains exports or by the LIB tool. For information on using LIB to create standard or import libraries, see LIB Reference. For details on using LINK to create an import library, see the /DLL option.

A library is specified to LINK as either a file name argument or a default library. LINK resolves external references by searching first in libraries specified on the command line, then in default libraries specified with the /DEFAULTLIB option, and then in default libraries named in `.obj` files. If a path is specified with the library name, LINK looks for the library in that directory. If no path is specified, LINK looks first in the directory that LINK is running from, and then in any directories specified in the `LIB` environment variable.

To add `.lib` files as linker input in the development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Choose the **Configuration Properties** > **Linker** > **Input** property page.
3. Modify the **Additional Dependencies** property to add the `.lib` files.
4. Choose **OK** or **Apply** to save your changes.

To programmatically add `.lib` files as linker input

- See AdditionalDependencies.

Example

The following sample shows how to build and use a `.lib` file.

First, build the `.lib` file:

C++

```
// lib_link_input_1.cpp
// compile by using: cl /LD lib_link_input_1.cpp
__declspec(dllexport) int Test() {
   return 213;
}
```

And then, compile this sample by using the `.lib` file you just created:

C++

```
// lib_link_input_2.cpp
// compile by using: cl /EHsc lib_link_input_1.lib lib_link_input_2.cpp
__declspec(dllimport) int Test();
#include <iostream>
int main() {
   std::cout << Test() << std::endl;
}
```

Output

```
213
```

See also

LINK input files

MSVC linker options

.Exp Files as Linker Input

Article • 08/03/2021

Export (.exp) files contain information about exported functions and data items. When LIB creates an import library, it also creates an .exp file. You use the .exp file when you link a program that both exports to and imports from another program, either directly or indirectly. If you link with an .exp file, LINK does not produce an import library, because it assumes that LIB already created one. For details about .exp files and import libraries, see Working with Import Libraries and Export Files.

See also

LINK Input Files
MSVC Linker Options

.Def Files as Linker Input

Article • 08/03/2021

See Module-definition (.def) files for more information. Use the /DEF option to specify the .def file name.

See also

LINK Input Files
MSVC Linker Options

.Pdb Files as Linker Input

Article • 08/03/2021

Object (.obj) files compiled using the /Zi option contain the name of a program database (PDB). You do not specify the object's PDB file name to the linker; LINK uses the embedded name to find the PDB if it is needed. This also applies to debuggable objects contained in a library; the PDB for a debuggable library must be available to the linker along with the library.

LINK also uses a PDB to hold debugging information for the .exe file or the .dll file. The program's PDB is both an output file and an input file, because LINK updates the PDB when it rebuilds the program.

See also

LINK Input Files
MSVC Linker Options

.Res Files as Linker Input

Article • 08/03/2021

You can specify a .res file when linking a program. The .res file is created by the resource compiler (RC). LINK automatically converts .res files to COFF. The CVTRES.exe tool must be in the same directory as LINK.exe or in a directory specified in the PATH environment variable.

See also

LINK Input Files
MSVC Linker Options

.Exe Files as Linker Input

Article • 08/03/2021

The MS-DOS Stub File Name (/STUB) option specifies the name of an .exe file that runs with MS-DOS. LINK examines the specified file to be sure that it is a valid MS-DOS program.

See also

LINK Input Files
MSVC Linker Options

.Txt Files as Linker Input

Article • 08/03/2021

LINK expects various text files as additional input. The command-file specifier (@) and the Base Address (/BASE), /DEF, and /ORDER options all specify text files. These files can have any extension, not just .txt.

See also

LINK Input Files
MSVC Linker Options

`.ilk` files as linker input

Article • 09/08/2022

The linker creates and uses a `.ilk` database file for incremental link information.

Remarks

When linking incrementally, LINK updates the `.ilk` status file that it created during the first incremental link. This file has the same base name as the target EXE or DLL file, and it has the extension `.ilk`. During subsequent incremental links, LINK updates the `.ilk` file. If the `.ilk` file is missing, LINK performs a full link and creates a new `.ilk` file. If the `.ilk` file is unusable, LINK performs a non-incremental link. For more information about incremental linking, see the /INCREMENTAL (Link incrementally) linker option. For information about how to specify the name and location of the file, see /ILK (Name incremental database file).

See also

LINK input files
MSVC linker options

LINK Output

Article • 08/03/2021

Link output includes .exe files, DLLs, mapfiles, and messages.

Output Files

The default output file from LINK is an .exe file. If the /DLL option is specified, LINK builds a .dll file. You can control the output file name with the Output File Name (/OUT) option.

In incremental mode, LINK creates an .ilk file to hold status information for later incremental builds of the program. For details about .ilk files, see .ilk Files. For more information about incremental linking, see the Link Incrementally (/INCREMENTAL) option.

When LINK creates a program that contains exports (usually a DLL), it also builds a .lib file, unless an .exp file was used in the build. You can control the import library file name with the /IMPLIB option.

If the Generate Mapfile (/MAP) option is specified, LINK creates a mapfile.

If the Generate Debug Info (/DEBUG) option is specified, LINK creates a PDB to contain debugging information for the program.

Other Output

When you type `link` without any other command-line input, LINK displays a usage statement that summarizes its options.

LINK displays a copyright and version message and echoes command-file input, unless the Suppress Startup Banner (/NOLOGO) option is used.

You can use the Print Progress Messages (/VERBOSE) option to display additional details about the build.

LINK issues error and warning messages in the form LNK*nnnn*. This error prefix and range of numbers are also used by LIB, DUMPBIN, and EDITBIN.

See also

MSVC linker reference

MSVC Linker Options

Reserved words

Article • 08/03/2021

The following words are reserved by the linker. These names can be used as arguments in module-definition statements only if the name is enclosed in double quotation marks ("").

`APPLOADER`[1]

`BASE`

`CODE`

`CONFORMING`

`DATA`

`DESCRIPTION`

`DEV386`

`DISCARDABLE`

`DYNAMIC`

`EXECUTE-ONLY`

`EXECUTEONLY`

`EXECUTEREAD`

`EXETYPE`

`EXPORTS`

`FIXED`[1]

`FUNCTIONS`[2]

`HEAPSIZE`

`IMPORTS`

`IMPURE`[1]

`INCLUDE`[2]

`INITINSTANCE`[2]

`IOPL`

`LIBRARY`[1]

`LOADONCALL`[1]

`LONGNAMES`[2]

`MOVABLE`[1]

`MOVEABLE`[1]

`MULTIPLE`

`NAME`

`NEWFILES`[2]

`NODATA`[1]

`NOIOPL`[1]

`NONAME`

`NONCONFORMING`[1]

`NONDISCARDABLE`

`NONE`

`NONSHARED`

`NOTWINDOWCOMPAT`[1]

`OBJECTS`

`OLD`[1]

`PRELOAD`

`PRIVATE`

`PROTMODE`[2]

`PURE`[1]

`READONLY`

`READWRITE`

`REALMODE`[1]

`RESIDENT`

`RESIDENTNAME`[1]

`SECTIONS`

`SEGMENTS`

`SHARED`

`SINGLE`

`STACKSIZE`

`STUB`

`VERSION`

`WINDOWAPI`

`WINDOWCOMPAT`

`WINDOWS`

[1] The linker emits a warning ("ignored") when it encounters this term. However, the word is still reserved.

[2] The linker ignores this word but emits no warning.

See also

- MSVC linker reference
- MSVC Linker Options

`@` (Specify a linker response file)

Article • 05/18/2022

Specifies a linker response file.

Syntax

> `@` *`response_file`*

Arguments

`response_file`

A text file that contains linker commands.

Remarks

For more information, see @ (Specify a compiler response file).

To set this linker option in the Visual Studio development environment

- This linker option isn't available from the Visual Studio development environment.

To set this linker option programmatically

- This linker option can't be changed programmatically.

See also

MSVC linker reference
MSVC linker options

`/ALIGN` (Section alignment)

Article • 05/18/2022

Specify the alignment of each section within the executable image.

Syntax

> `/ALIGN`[`:`*`number`*]

Arguments

`number`
The alignment value in bytes.

Remarks

The `/ALIGN` linker option specifies the alignment of each section within the linear address space of the program. The *`number`* argument is in bytes and must be a power of two. The default is 4K (4096). The linker issues a warning if the alignment produces an invalid image.

Unless you're writing an application such as a device driver, you shouldn't need to modify the alignment.

It's possible to modify the alignment of a particular section with the *`align`* parameter to the /SECTION option.

The alignment value that you specify can't be smaller than the largest section alignment.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Choose the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter the option in the **Additional Options** box. Choose **OK** or **Apply** to apply the change.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options

`/ALLOWBIND` (Prevent DLL binding)

Article • 05/18/2022

Set a flag to disallow DLL binding.

Syntax

> `/ALLOWBIND`[`:NO`]

Remarks

The `/ALLOWBIND:NO` linker option sets a bit in a DLL's header that indicates to Bind.exe that the image can't be bound. You may not want a DLL to be bound if it's been digitally signed (binding invalidates the signature).

You can edit an existing DLL for `/ALLOWBIND` functionality with the /ALLOWBIND option of the EDITBIN utility.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter `/ALLOWBIND:NO` into **Additional Options**. Choose **OK** or **Apply** to apply the change.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options
BindImage function
BindImageEx function

`/ALLOWISOLATION` (Manifest lookup)

Article • 05/18/2022

Specifies behavior for manifest lookup.

Syntax

> `/ALLOWISOLATION`[`:NO`]

Remarks

The `/ALLOWISOLATION:NO` linker option indicates DLLs are loaded as if there was no manifest and causes the linker to set the `IMAGE_DLLCHARACTERISTICS_NO_ISOLATION` bit in the optional header's `DllCharacteristics` field.

`/ALLOWISOLATION` causes the operating system to do manifest lookups and loads.

`/ALLOWISOLATION` is the default.

When isolation is disabled for an executable, the Windows loader won't attempt to find an application manifest for the newly created process. The new process won't have a default activation context, even if there's a manifest inside the executable or placed in the same directory as the executable with name *`<executable-name>.exe.manifest`*.

For more information, see Manifest files reference.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Manifest File** property page.
3. Modify the **Allow Isolation** property.

See also

MSVC linker reference
MSVC linker options

`/APPCONTAINER` (Microsoft Store app)

Article • 05/18/2022

Specifies whether the linker creates an executable image that must run in an AppContainer environment.

Syntax

> `/APPCONTAINER`[`:NO`]

Remarks

The `/APPCONTAINER` linker option modifies an executable to indicate whether the app must run in the AppContainer process-isolation environment. Specify `/APPCONTAINER` for an app that must run in the AppContainer environment—for example, a Universal Windows Platform (UWP) or Windows Phone 8.x app. The option is set automatically in Visual Studio when you create a Universal Windows app from a template. For a desktop app, specify `/APPCONTAINER:NO` or just omit the option. By default, `/APPCONTAINER` is off.

The `/APPCONTAINER` option was introduced in Windows 8.

To set this linker option in Visual Studio

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. In **Additional Options**, enter *`/APPCONTAINER`* or *`/APPCONTAINER:NO`*.

See also

MSVC linker reference
MSVC linker options

/ARM64XFUNCTIONPADMINX64 (Minimum x64 function padding)

Article • 01/10/2024

Specifies the minimum number of bytes of padding between x64 functions in ARM64X images.

Syntax

```
/ARM64XFUNCTIONPADMINX64:[number]
```

Arguments

number
The minimum number of bytes of padding between x64 functions.

Remarks

This switch ensures that there is at least as much padding between X64 functions in an ARM64X image as specified. There may be more padding to meet architecture alignment requirements.

This flag is available in Visual Studio 17.8 and later.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Modify the **Additional Options** property to include **/ARM64XFUNCTIONPADMINX64:**`number`, where `number` is the minimum number of bytes of padding to put between x64 functions, and then choose **OK**.

To set this linker option programmatically

- See AdditionalOptions.

See also

/FUNCTIONPADMIN (Create hotpatchable image)
/NOFUNCTIONPADSECTION
MSVC Linker Options
MSVC linker reference

`/ASSEMBLYDEBUG` (Add DebuggableAttribute)

Article • 05/18/2022

Specify whether to emit the `DebuggableAttribute` attribute with debug information tracking and to disable JIT optimizations.

Syntax

> `/ASSEMBLYDEBUG`[`:DISABLE`]

Remarks

The `/ASSEMBLYDEBUG` linker option emits the `DebuggableAttribute` attribute with debug information tracking and disables JIT optimizations. This option is the same as specifying the following attribute in source:

```
C++

[assembly:Debuggable(true, true)];   // same as /ASSEMBLYDEBUG
```

`/ASSEMBLYDEBUG:DISABLE` emits the `DebuggableAttribute` attribute but disables the tracking of debug information and enables JIT optimizations. This option is the same as specifying the following attribute in source:

```
C++

[assembly:Debuggable(false, false)];   // same as /ASSEMBLYDEBUG:DISABLE
```

By default, the linker doesn't emit the `DebuggableAttribute` attribute.

`DebuggableAttribute` can also be added to an assembly directly in source code. For example:

```
C++

[assembly:Debuggable(true, true)];   // same as /ASSEMBLYDEBUG
```

You must explicitly specify that a managed image is debuggable. The /Zi option alone is insufficient.

Other linker options that affect assembly generation are:

- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /DELAYSIGN
- /KEYCONTAINER
- /KEYFILE
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Debug** property page.
3. Modify the **Debuggable Assembly** property.

To set this linker option programmatically

- See AssemblyDebug.

See also

MSVC linker reference
MSVC linker options

`/ASSEMBLYLINKRESOURCE` (Link to .NET Framework resource)

Article • 05/18/2022

Create a link to a .NET Framework resource in the output file.

Syntax

> **`/ASSEMBLYLINKRESOURCE:`***`filename`*

Arguments

`filename` The .NET Framework resource file to link from the assembly.

Remarks

The **`/ASSEMBLYLINKRESOURCE`** linker option creates a link to a .NET Framework resource in the output file. The resource file isn't placed in the output file. Use the /ASSEMBLYRESOURCE option to embed a resource file in the output file.

Linked resources are public in the assembly when created with the linker.

`/ASSEMBLYLINKRESOURCE` requires the /clr compiler option. The /LN or /NOASSEMBLY options aren't allowed with **`/ASSEMBLYLINKRESOURCE`**.

If *`filename`* is a .NET Framework resource file that's created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the `System.Resources` namespace. For more information, see System.Resources.ResourceManager. For all other resources, use the `GetManifestResource*` methods in the `System.Reflection.Assembly` class to access the resource at run time.

`filename` can have any file format. For example, you may want to make a native DLL part of the assembly. Then it can be installed into the Global Assembly Cache and accessed from managed code in the assembly.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG

- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /DELAYSIGN
- /KEYCONTAINER
- /KEYFILE
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter the option in **Additional Options**. Choose **OK** or **Apply** to apply the change.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options

`/ASSEMBLYMODULE` (Add an MSIL module to the assembly)

Article • 05/18/2022

Syntax

> `/ASSEMBLYMODULE:`*`filename`*

Arguments

`filename`
The module you want to include in this assembly.

Remarks

The `/ASSEMBLYMODULE` linker option allows you to add a module reference to an assembly. Type information in the module won't be available to the assembly program that added the module reference. However, type information in the module will be available to any program that references the assembly.

Use #using to both add a module reference to an assembly and make the module's type information available to the assembly program.

For example, consider the following scenario:

1. Create a module with /LN.
2. Use `/ASSEMBLYMODULE` in a different project to include the module in the current compilation, which creates an assembly. This project won't reference the module with `#using`.
3. Any project that references this assembly can now also use types from the module.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYRESOURCE

- /DELAYSIGN
- /NOASSEMBLY
- /KEYFILE
- /KEYCONTAINER

The MSVC linker accepts `.netmodule` files as input and the output file produced by the linker will be an assembly or `.netmodule` file with no run-time dependence on any of the `.netmodule` files that were input to the linker. For more information, see .netmodule files as linker input.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Input** property page.
3. Modify the **Add Module to Assembly** property.

To set this linker option programmatically

- See AddModuleNamesToAssembly.

See also

MSVC linker reference
MSVC linker options

`/ASSEMBLYRESOURCE` (Embed a managed resource)

Article • 05/18/2022

Embeds a managed resource in an assembly.

Syntax

> `/ASSEMBLYRESOURCE:`*`filename`*[`,`[*`name`*][`,PRIVATE`]]

Arguments

`filename`
The managed resource you want to embed in this assembly.

`name`
Optional. The logical name for the resource; the name used to load the resource. The default is the name of the file.

Optionally, you can use `PRIVATE` to specify if the file should be private in the assembly manifest. By default, *`name`* is public in the assembly.

Remarks

Use the `/ASSEMBLYRESOURCE` linker option to embed a resource in an assembly.

Resources are public in the assembly when created with the linker. The linker doesn't allow you to rename the resource in the assembly.

If *`filename`* is a .NET Framework resource (*`.resources`*) file created, for example, by the Resource file generator (Resgen.exe) or in the development environment, it can be accessed with members in the `System.Resources` namespace. For more information, see System.Resources.ResourceManager. For all other resources, use the `GetManifestResource*` methods in the `System.Reflection.Assembly` class to access the resource at run time.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG

- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /DELAYSIGN
- /KEYFILE
- /KEYCONTAINER
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Input** property page.
3. Modify the **Embed Managed Resource File** property.

To set this linker option programmatically

1. See EmbedManagedResourceFile.

See also

MSVC linker reference
MSVC linker options

`/BASE` (Base address)

Article • 05/18/2022

Specifies the base address for a program.

Syntax

> `/BASE:`{*`address`*[`,`*`size`*] | `@`*`filename`*`,`*`key`*}

Remarks

> **Note**
>
> For security reasons, Microsoft recommends you use the /DYNAMICBASE option instead of specifying base addresses for your executables. `/DYNAMICBASE` generates an executable image that can be randomly rebased at load time by using the address space layout randomization (ASLR) feature of Windows. The `/DYNAMICBASE` option is on by default.

The `/BASE` linker option sets a base address for the program. It overrides the default location for an EXE or DLL file. The default base address for an EXE file is 0x400000 for 32-bit images or 0x140000000 for 64-bit images. For a DLL, the default base address is 0x10000000 for 32-bit images or 0x180000000 for 64-bit images. On operating systems that don't support address space layout randomization (ASLR), or when the `/DYNAMICBASE:NO` option was set, the operating system first attempts to load a program at its specified or default base address. If insufficient space is available there, the system relocates the program. To prevent relocation, use the /FIXED option.

The linker issues an error if *`address`* isn't a multiple of 64K. You can optionally specify the size of the program. The linker issues a warning if the program can't fit in the size you specified.

On the command line, another way to specify the base address is by using a *base address response file*. A base address response file is a text file that contains the base addresses and optional sizes of all the DLLs your program uses, and a unique text key for each base address. To specify a base address by using a response file, use an at sign (`@`) followed by the name of the response file, *`filename`*, followed by a comma, then the *`key`* value for the base address to use in the file. The linker looks for *`filename`* in either

the specified path, or if no path is specified, in the directories specified in the `LIB` environment variable. Each line in *`filename`* represents one DLL and has the following syntax:

> *`key`* *`address`* [*`size`*] **`;`** *`comment`*

The *`key`* is a string of alphanumeric characters and isn't case sensitive. It's usually the name of a DLL, but that's not required. The *`key`* is followed by a base *`address`* in C-language, hexadecimal, or decimal notation and an optional maximum *`size`*. All three arguments are separated by spaces or tabs. The linker issues a warning if the specified *`size`* is less than the virtual address space required by the program. A *`comment`* is specified by a semicolon (**`;`**) and can be on the same or a separate line. The linker ignores all text from the semicolon to the end of the line. This example shows part of such a file:

txt

```
main    0x00010000    0x08000000    ; for PROJECT.exe
one     0x28000000    0x00100000    ; for DLLONE.DLL
two     0x28100000    0x00300000    ; for DLLTWO.DLL
```

If the file that contains these lines is called DLLS.txt, the following example command applies this information:

Windows Command Prompt

```
link dlltwo.obj /dll /base:@dlls.txt,two
```

Another way to set the base address is by using the *`BASE`* argument in a NAME or LIBRARY statement. The **`/BASE`** and /DLL options together are equivalent to the **`LIBRARY`** statement.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Base Address** property.

To set this linker option programmatically

- See BaseAddress.

See also

MSVC linker reference
MSVC linker options

`/CETCOMPAT` (CET Shadow Stack compatible)

Article • 09/22/2022

Specifies whether the linker marks an executable image as compatible with Control-flow Enforcement Technology (CET) Shadow Stack.

Syntax

> `/CETCOMPAT`
>
> `/CETCOMPAT:NO`

Arguments

`NO`

Specifies that the executable shouldn't be marked compatible with CET Shadow Stack.

Remarks

Control-flow Enforcement Technology (CET) Shadow Stack is a computer processor feature. It provides capabilities to defend against return-oriented programming (ROP) based malware attacks. For more information, see A Technical Look at Intel's Control-flow Enforcement Technology.

The `/CETCOMPAT` linker option tells the linker to mark the binary as CET Shadow Stack-compatible. `/CETCOMPAT:NO` marks the binary as not compatible with CET Shadow Stack. If both options are specified on the command line, the last one specified is used. This switch is currently only applicable to x86 and x64 architectures.

The `/CETCOMPAT` option is available beginning in Visual Studio 2019.

To set the `/CETCOMPAT` linker option in Visual Studio

Starting in Visual Studio 2019 version 16.7:

1. Open the **Property Pages** dialog box for the project. For more information, see Set compiler and build properties.

2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.

3. Select the **CET Shadow Stack Compatible** property.

4. In the dropdown control, choose `Yes (/CETCOMPAT)` to mark the binary as CET Shadow Stack compatible, or `No (/CETCOMPAT:NO)` to mark it as non-compatible.

In previous versions of Visual Studio 2019:

1. Open the **Property Pages** dialog box for the project. For more information, see Set compiler and build properties.

2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.

3. In the **Additional Options** edit control, add `/CETCOMPAT` to mark the binary as CET Shadow Stack compatible, or `/CETCOMPAT:NO` to explicitly mark it as non-compatible.

To set this linker option programmatically

This option doesn't have a programmatic equivalent.

See also

Linker options

/CGTHREADS (Compiler Threads)

Article • 08/03/2021

Sets the number of cl.exe threads to use for optimization and code generation when link-time code generation is specified.

Syntax

```
/CGTHREADS:[1-8]
```

Arguments

number
The maximum number of threads for cl.exe to use, in the range 1 to 8.

Remarks

The **/CGTHREADS** option specifies the maximum number of threads cl.exe uses in parallel for the optimization and code-generation phases of compilation when link-time code generation (/LTCG) is specified. By default, cl.exe uses four threads, as if **/CGTHREADS:4** were specified. If more processor cores are available, a larger `number` value can improve build times.

Multiple levels of parallelism can be specified for a build. The msbuild.exe switch **/maxcpucount** specifies the number of MSBuild processes that can be run in parallel. The /MP (Build with Multiple Processes) compiler flag specifies the number of cl.exe processes that simultaneously compile the source files. The /cgthreads compiler option specifies the number of threads used by each cl.exe process. Because the processor can only run as many threads at the same time as there are processor cores, it's not useful to specify larger values for all of these options at the same time, and it can be counterproductive. For more information about how to build projects in parallel, see Building Multiple Projects in Parallel.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Modify the **Additional Options** property to include **/CGTHREADS:**`number`, where `number` is a value from 1 to 8, and then choose **OK**.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC Linker Options
MSVC linker reference

/CLRIMAGETYPE (Specify Type of CLR Image)

Article • 08/03/2021

Set the CLR image type in the linked image.

Syntax

> **/CLRIMAGETYPE:{IJW|PURE|SAFE|SAFE32BITPREFERRED}**

Remarks

The linker accepts native objects and also MSIL objects that are compiled by using /clr. The **/clr:pure** and **/clr:safe** compiler options were deprecated in Visual Studio 2015 and are unsupported in Visual Studio 2017 and later. When mixed objects in the same build are passed, the verifiability of the resulting output file is, by default, equal to the lowest level of verifiability of the input modules. For example, if you pass a native image and a mixed mode image (compiled by using **/clr**), the resulting image will be a mixed mode image.

You can use **/CLRIMAGETYPE** to specify a lower level of verifiability, if that is what you need.

For information about how to determine the CLR image type of a file, see /CLRHEADER.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **CLR Image Type** property.

To set this linker option programmatically

1. See CLRImageType.

See also

- MSVC linker reference
- MSVC Linker Options

`/CLRSUPPORTLASTERROR` (Preserve Last Error Code for PInvoke Calls)

Article • 09/22/2022

`/CLRSUPPORTLASTERROR`, which is on by default, preserves the last error code of functions called through the P/Invoke mechanism, which allows you to call native functions in DLLS, from code compiled with `/clr`.

Syntax

> `/CLRSUPPORTLASTERROR`
>
> `/CLRSUPPORTLASTERROR:NO`
>
> `/CLRSUPPORTLASTERROR:SYSTEMDLL`

Remarks

Preserving the last error code implies a decrease in performance. If you don't want to incur the performance cost of preserving the last error code, link by using `/CLRSUPPORTLASTERROR:NO`.

You can minimize the performance penalty by linking with `/CLRSUPPORTLASTERROR:SYSTEMDLL`, which only preserves the last error code for functions in system DLLs.

> ⓘ Note
>
> Preserving the last error isn't supported for unmanaged functions that are consumed by CLR code in the same module.

- For more information, see /clr (Common Language Runtime Compilation).

To set this linker option in the Visual Studio development environment

1. Open the **Property Pages** dialog box for the project. For more information, see Set compiler and build properties.

2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.

3. Modify the **Preserve Last Error Code for PInvoke Calls** property. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

Examples

The following sample defines a native DLL with one exported function that modifies last error.

C++

```
// CLRSUPPORTLASTERROR_dll.cpp
// compile with: /LD
#include <windows.h>
#include <math.h>

#pragma unmanaged
__declspec(dllexport) double MySqrt(__int64 n) {
   SetLastError(DWORD(-1));
   return sqrt(double(n));
}
```

The following sample consumes the DLL, demonstrating how to use `/CLRSUPPORTLASTERROR`.

C++

```
// CLRSUPPORTLASTERROR_client.cpp
// compile with: /clr CLRSUPPORTLASTERROR_dll.lib /link
/clrsupportlasterror:systemdll
// processor: x86
#include <windows.h>
#include <wininet.h>
#include <stdio.h>
#include <math.h>

#pragma comment(lib, "wininet.lib")

double MySqrt(__int64 n);

#pragma managed
int main() {
   double   d = 0.0;
   __int64 n = 65;
   HANDLE  hGroup = NULL;
```

```
    GROUPID groupID;
    DWORD   dwSet = 127, dwGet = 37;

    SetLastError(dwSet);
    d = MySqrt(n);
    dwGet = GetLastError();

    if (dwGet == DWORD(-1))
       printf_s("GetLastError for application call succeeded (%d).\n",
              dwGet);
    else
       printf_s("GetLastError for application call failed (%d).\n",
              dwGet);

    hGroup = FindFirstUrlCacheGroup(0, CACHEGROUP_SEARCH_ALL,
                            0, 0, &groupID, 0);
    dwGet = GetLastError();
    if (dwGet == 183)
       printf_s("GetLastError for system call succeeded (%d).\n",
              dwGet);
    else
       printf_s("GetLastError for system call failed (%d).\n",
              dwGet);
}
```

Output

```
GetLastError for application call failed (127).
GetLastError for system call succeeded (183).
```

See also

MSVC linker reference

MSVC linker options

/CLRTHREADATTRIBUTE (Set CLR Thread Attribute)

Article • 08/03/2021

Explicitly specify the threading attribute for the entry point of your CLR program.

Syntax

```
/CLRTHREADATTRIBUTE:{STA|MTA|NONE}
```

Parameters

MTA
Applies the MTAThreadAttribute attribute to the entry point of your program.

NONE
Same as not specifying /CLRTHREADATTRIBUTE. Lets the Common Language Runtime (CLR) set the default threading attribute.

STA
Applies the STAThreadAttribute attribute to the entry point of your program.

Remarks

Setting the thread attribute is only valid when building an .exe, as it affects the entry point of the main thread.

If you use the default entry point (main or wmain, for example) specify the threading model either by using /CLRTHREADATTRIBUTE or by placing the threading attribute (STAThreadAttribute or MTAThreadAttribute) on the default entry function.

If you use a non-default entry point, specify the threading model either by using /CLRTHREADATTRIBUTE or by placing the threading attribute on the non-default entry function, and then specify the non-default entry point with /ENTRY.

If the threading model specified in source code does not agree with the threading model specified with /CLRTHREADATTRIBUTE, the linker will ignore /CLRTHREADATTRIBUTE and apply the threading model specified in source code.

It will be necessary for you to use single-threading, for example, if your CLR program hosts a COM object that uses single-threading. If your CLR program uses multi-threading, it cannot host a COM object that uses single-threading.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **CLR Thread Attribute** property.

To set this linker option programmatically

1. See CLRThreadAttribute.

See also

MSVC linker reference
MSVC Linker Options

`/CLRUNMANAGEDCODECHECK` (Remove SuppressUnmanagedCodeSecurityAttribute)

Article • 03/02/2023

`/CLRUNMANAGEDCODECHECK` specifies that the linker doesn't apply SuppressUnmanagedCodeSecurityAttribute to linker-generated `PInvoke` calls from managed code into native DLLs.

Syntax

> `/CLRUNMANAGEDCODECHECK`
>
> `/CLRUNMANAGEDCODECHECK:NO`

Remarks

By default, the linker applies the `SuppressUnmanagedCodeSecurityAttribute` attribute to linker-generated `PInvoke` calls. When `/CLRUNMANAGEDCODECHECK` is in effect, `SuppressUnmanagedCodeSecurityAttribute` is removed. To explicitly apply the `SuppressUnmanagedCodeSecurityAttribute` attribute to linker-generated `PInvoke` calls, you can use `/CLRUNMANAGEDCODECHECK:NO`.

The linker only adds the attribute to objects that are compiled using `/clr` or `/clr:pure`. However, the `/clr:pure` compiler option is deprecated in Visual Studio 2015 and unsupported in Visual Studio 2017 and later.

A `PInvoke` call is generated by the linker when the linker can't find a managed symbol to satisfy a reference from a managed caller but can find a native symbol to satisfy that reference. For more information about `PInvoke`, see Calling Native Functions from Managed Code.

If you use AllowPartiallyTrustedCallersAttribute in your code, you should explicitly set `/CLRUNMANAGEDCODECHECK` to remove the `SuppressUnmanagedCodeSecurity` attribute. It's a potential security vulnerability if an image contains both the `SuppressUnmanagedCodeSecurity` and `AllowPartiallyTrustedCallers` attributes.

For more information about the implications of using `SuppressUnmanagedCodeSecurityAttribute`, see Secure Coding Guidelines for Unmanaged Code.

To set this linker option in the Visual Studio development environment

1. Open the **Property Pages** dialog box for the project. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **CLR Unmanaged Code Check** property.

To set this linker option programmatically

1. See CLRUnmanagedCodeCheck.

See also

MSVC linker reference
MSVC linker options

`/DEBUG` (Generate debug info)

Article • 10/27/2023

The `/DEBUG` linker option creates a debugging information file for the executable.

Syntax

> `/DEBUG`[`:`{`FASTLINK`|`FULL`|`NONE`}]

Remarks

The `/DEBUG` option puts the debugging information from linked object and library files into a program database (PDB) file. It updates the PDB during subsequent builds of the program.

An executable (an EXE or DLL file) created for debugging contains the name and path of the corresponding PDB. The debugger reads the embedded name and uses the PDB when you debug the program. The linker uses the base name of the program and the extension `.pdb` to name the program database, and embeds the path where it was created. To override this default, set the /PDB option and specify a different file name.

The `/DEBUG:FASTLINK` option is available in Visual Studio 2017 and later. This option generates a limited PDB that indexes into the debug information in the object files and libraries used to build the executable instead of making a full copy. You can only use this limited PDB to debug from the computer where the binary and its libraries were built. If you deploy the binary elsewhere, you may debug it remotely from the build computer, but not directly on the test computer. Despite its name, `/DEBUG:FASTLINK` is generally slower than `/DEBUG:FULL`, so this option is not recommended.

A `/DEBUG:FASTLINK` PDB can be converted to a full PDB that you can deploy to a test machine for local debugging. In Visual Studio, use the **Property Pages** dialog as described below to create a full PDB for the project or solution. In a developer command prompt, you can use the `mspdbcmf.exe` tool to create a full PDB.

The `/DEBUG:FULL` option moves all private symbol information from individual compilation products (object files and libraries) into a single PDB, and can be the most time-consuming part of the link. However, the full PDB can be used to debug the executable when no other build products are available, such as when the executable is deployed.

The `/DEBUG:NONE` option doesn't generate a PDB.

Specifying `/DEBUG` with no extra arguments is equivalent to specifying `/DEBUG:FULL`.

The compiler's /Z7 (C7 Compatible) option causes the compiler to leave the debugging information in the object (OBJ) files. You can also use the /Zi (Program Database) compiler option to store the debugging information in a PDB for the OBJ file. The linker looks for the object's PDB first in the absolute path written in the OBJ file, and then in the directory that contains the OBJ file. You can't specify an object's PDB file name or location to the linker.

/INCREMENTAL is implied when `/DEBUG` is specified.

`/DEBUG` changes the defaults for the /OPT option from `REF` to `NOREF` and from `ICF` to `NOICF`, so if you want the original defaults, you must explicitly specify `/OPT:REF` or `/OPT:ICF` after the `/DEBUG` option.

It isn't possible to create an EXE or DLL that contains debug information. Debug information is always placed in an OBJ or PDB file.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Linker** > **Debugging** property page.
3. Modify the **Generate Debug Info** property to enable or disable PDB generation. This property enables `/DEBUG:FASTLINK` by default in Visual Studio 2017 and later.
4. Modify the **Generate Full Program Database File** property to enable `/DEBUG:FULL` for full PDB generation for every incremental build.

To set this linker option programmatically

1. See GenerateDebugInformation.

See also

MSVC linker reference
MSVC linker options

/DEBUGTYPE (Debug Info Options)

Article • 08/03/2021

The /DEBUGTYPE option specifies the types of debugging information generated by the /DEBUG option.

```
/DEBUGTYPE:[CV | PDATA | FIXUP]
```

Arguments

CV
Tells the linker to emit debug information for symbols, line numbers, and other object compilation information in the PDB file. By default, this option is enabled when **/DEBUG** is specified and **/DEBUGTYPE** is not specified.

PDATA
Tells the linker to add .pdata and .xdata entries to the debug stream information in the PDB file. By default, this option is enabled when both the **/DEBUG** and **/DRIVER** options are specified. If **/DEBUGTYPE:PDATA** is specified by itself, the linker automatically includes debugging symbols in the PDB file. If **/DEBUGTYPE:PDATA,FIXUP** is specified, the linker does not include debugging symbols in the PDB file.

FIXUP
Tells the linker to add relocation table entries to the debug stream information in the PDB file. By default, this option is enabled when both the **/DEBUG** and **/PROFILE** options are specified. If **/DEBUGTYPE:FIXUP** or **/DEBUGTYPE:FIXUP,PDATA** is specified, the linker does not include debugging symbols in the PDB file.

Arguments to **/DEBUGTYPE** may be combined in any order by separating them with a comma. The **/DEBUGTYPE** option and its arguments are not case sensitive.

Remarks

Use the **/DEBUGTYPE** option to specify inclusion of relocation table data or .pdata and .xdata header information in the debugging stream. This causes the linker to include information about user-mode code that is visible in a kernel debugger when breaking in kernel-mode code. To make debugging symbols available when **FIXUP** is specified, include the **CV** argument.

To debug code in user mode, which is typical for applications, the **/DEBUGTYPE** option isn't needed. By default, the compiler switches that specify debugging output (/Z7, /Zi, /ZI) emit all the information needed by the Visual Studio debugger. Use **/DEBUGTYPE:PDATA** or **/DEBUGTYPE:CV,PDATA,FIXUP** to debug code that combines user-mode and kernel-mode components, such as a configuration app for a device driver. For more information about kernel mode debuggers, see Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)

See also

/DEBUG (Generate Debug Info)
/DRIVER (Windows NT Kernel Mode Driver)
/PROFILE (Performance Tools Profiler)
Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)

`/DEF` (Specify module-definition file)

Article • 09/22/2022

Specifies a module-definition file to the linker.

Syntax

> `/DEF:`*`filename`*

Arguments

`filename`
The name of a module-definition file (*`.def`*) to be passed to the linker.

Remarks

The `/DEF` linker option passes a module-definition file (*`.def`*) to the linker. Only one *`.def`* file can be specified to LINK. For details about *`.def`* files, see Module-definition files.

To specify a *`.def`* file from within the development environment, add it to the project along with your other source files and then specify the file in the project's **Property Pages** dialog.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **Input** property page.
3. Modify the **Module Definition File** property. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See ModuleDefinitionFile.

See also

MSVC linker reference
MSVC Linker Options

/DEFAULTLIB (Specify Default Library)

Article • 08/03/2021

Specify a default library to search to resolve external references.

Syntax

> **/DEFAULTLIB**:*library*

Arguments

library
The name of a library to search when resolving external references.

Remarks

The **/DEFAULTLIB** option adds one *library* to the list of libraries that LINK searches when resolving references. A library specified with **/DEFAULTLIB** is searched after libraries specified explicitly on the command line and before default libraries named in .obj files.

When used without arguments, the /NODEFAULTLIB (Ignore All Default Libraries) option overrides all **/DEFAULTLIB**:*library* options. The **/NODEFAULTLIB**:*library* option overrides **/DEFAULTLIB**:*library* when the same *library* name is specified in both.

To set this linker option in the Visual Studio development environment

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. In **Additional Options**, enter a **/DEFAULTLIB**:*library* option for each library to search. Choose **OK** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

- MSVC linker reference
- MSVC Linker Options

`/DELAY` (Delay load import settings)

Article • 09/22/2022

Linker options to control delayed loading of DLLs at runtime.

Syntax

> `/DELAY:UNLOAD`
>
> `/DELAY:NOBIND`

Remarks

The `/DELAY` option controls delayed loading of DLLs:

- The `/DELAY:UNLOAD` qualifier tells the delay-load helper function to support explicit unloading of the DLL. The Import Address Table (IAT) is reset to its original form, invalidating IAT pointers and causing them to be overwritten.

 If you don't select `/DELAY:UNLOAD`, any call to __FUnloadDelayLoadedDLL will fail.

- The `/DELAY:NOBIND` qualifier tells the linker not to include a bindable IAT in the final image. The default is to create the bindable IAT for delay-loaded DLLs. The resulting image can't be statically bound. (Images with bindable IATs may be statically bound before execution.) For more information, see /BIND.

 If the DLL is bound, the helper function attempts to use the bound information instead of calling GetProcAddress on each of the referenced imports. If either the timestamp or the preferred address doesn't match the ones in the loaded DLL, the helper function assumes the bound IAT is out of date. It continues as if the bound IAT doesn't exist.

 `/DELAY:NOBIND` causes your program image to be larger, but can speed load time of the DLL. If you never intend to bind the DLL, `/DELAY:NOBIND` prevents the bound IAT from being generated.

To specify DLLs to delay load, use the /DELAYLOAD option.

To set this linker option in the Visual Studio development environment

1. Open the **Property Pages** dialog box for the project. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Unload delay loaded DLL** property or the **Unbind delay loaded DLL** property. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options

/DELAYLOAD (Delay Load Import)

Article • 08/03/2021

> **/DELAYLOAD:***dllname*

Parameters

dllname
The name of a DLL that you want to delay load.

Remarks

The /DELAYLOAD option causes the DLL that's specified by `dllname` to be loaded only on the first call by the program to a function in that DLL. For more information, see Linker Support for Delay-Loaded DLLs. You can use this option as many times as necessary to specify as many DLLs as you choose. You must use Delayimp.lib when you link your program, or you can implement your own delay-load helper function.

The /DELAY option specifies binding and loading options for each delay-loaded DLL.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. In the **Linker** folder, select the **Input** property page.
3. Modify the **Delay Loaded DLLs** property.

To set this linker option programmatically

- See DelayLoadDLLs.

See also

MSVC linker reference
MSVC Linker Options

/DELAYSIGN (Partially Sign an Assembly)

Article • 08/03/2021

```
/DELAYSIGN[:NO]
```

Arguments

NO
Specifies that the assembly should not be partially signed.

Remarks

Use **/DELAYSIGN** if you only want to place the public key in the assembly. The default is **/DELAYSIGN:NO**.

The **/DELAYSIGN** option has no effect unless used with /KEYFILE or /KEYCONTAINER.

When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. The resulting digital signature is stored in the file that contains the manifest. When an assembly is delay signed, the linker does not compute and store the signature, but reserves space in the file so the signature can be added later.

For example, using **/DELAYSIGN** allows a tester to put the assembly in the global cache. After testing, you can fully sign the assembly by placing the private key in the assembly.

See Strong Name Assemblies (Assembly Signing) (C++/CLI) and Delay Signing an Assembly for more information on signing an assembly.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE

- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Click the **Linker** folder.
3. Click the **Command Line** property page.
4. Type the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC Linker Options

/DEPENDENTLOADFLAG (Set default dependent load flags)

Article • 10/06/2021

Sets the default load flags used when the operating system resolves the statically linked imports of a module.

Syntax

> **/DEPENDENTLOADFLAG**[:*load_flags*]

Arguments

load_flags
An optional integer value that specifies the load flags to apply when resolving statically linked import dependencies of the module. The default value is 0. For a list of supported flag values, see the `LOAD_LIBRARY_SEARCH_*` entries in LoadLibraryEx.

Remarks

When the operating system resolves the statically linked imports of a module, it uses the default search order. Use the **/DEPENDENTLOADFLAG** option to specify a *load_flags* value that changes the search path used to resolve these imports. On supported operating systems, it changes the static import resolution search order, similar to what LoadLibraryEx does when using `LOAD_LIBRARY_SEARCH` parameters. For information on the search order set by *load_flags*, see Search order using LOAD_LIBRARY_SEARCH flags.

This flag can be used to make one DLL planting attack vector more difficult. For example, consider an app that has statically linked a DLL:

- An attacker could plant a DLL with the same name earlier in the import resolution search path, such as the application directory. Protected directories are more difficult - but not impossible - for an attacker to change.
- If the DLL is missing from the application, %windows%\system32, and %windows% directories, import resolution falls through to the current directory. An attacker could plant a DLL there.

In both cases, if you specify the link option `/DEPENDENTLOADFLAG:0x800` (the value of the flag `LOAD_LIBRARY_SEARCH_SYSTEM32`), then the module search path is limited to the %windows%\system32 directory. It offers some protection from planting attacks on the other directories. For more information, see Dynamic-Link Library Security.

To see the value set by the **/DEPENDENTLOADFLAG** option in any DLL, use the DUMPBIN command with the /LOADCONFIG option.

The **/DEPENDENTLOADFLAG** option is new in Visual Studio 2017. It applies only to apps running on Windows 10 RS1 and later Windows versions. This option is ignored by other operating systems that run the app.

To set the DEPENDENTLOADFLAG linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter the option in **Additional Options**.

To set this linker option programmatically

- See AdditionalOptions.

See also

- MSVC linker reference
- MSVC linker options
- Link an executable to a DLL implicitly
- Determine which linking method to use
- LoadLibraryEx
- Dynamic-Link Library Search Order
- Dynamic-Link Library Security

/DLL (Build a DLL)

Article • 08/03/2021

```
/DLL
```

Remarks

The /DLL option builds a DLL as the main output file. A DLL usually contains exports that can be used by another program. There are three methods for specifying exports, listed in recommended order of use:

1. __declspec(dllexport) in the source code
2. An EXPORTS statement in a .def file
3. An /EXPORT specification in a LINK command

A program can use more than one method.

Another way to build a DLL is with the **LIBRARY** module-definition statement. The /BASE and /DLL options together are equivalent to the **LIBRARY** statement.

Do not specify this option within the development environment; this option is for use only on the command line. This option is set when you create a DLL project with an Application Wizard.

Note that if you create your import library in a preliminary step, before creating your .dll, you must pass the same set of object files when building the .dll, as you passed when building the import library.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Click the **Configuration Properties** folder.
3. Click the **General** property page.

4. Modify the **Configuration Type** property.

To set this linker option programmatically

- See ConfigurationType.

See also

MSVC linker reference
MSVC Linker Options

/DRIVER (Windows NT Kernel Mode Driver)

Article • 08/03/2021

> /DRIVER[:UPONLY |:WDM]

Remarks

Use the **/DRIVER** linker option to build a Windows NT kernel mode driver.

/DRIVER:UPONLY causes the linker to add the **IMAGE_FILE_UP_SYSTEM_ONLY** bit to the characteristics in the output header to specify that it is a uniprocessor (UP) driver. The operating system will refuse to load a UP driver on a multiprocessor (MP) system.

/DRIVER:WDM causes the linker to set the **IMAGE_DLLCHARACTERISTICS_WDM_DRIVER** bit in the optional header's DllCharacteristics field.

If **/DRIVER** is not specified, these bits are not set by the linker.

If **/DRIVER** is specified:

- **/FIXED:NO** is in effect. For more information, see /FIXED (Fixed Base Address).
- The extension of the output file is set to .sys. Use **/OUT** to change the default filename and extension. For more information, see /OUT (Output File Name).

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Click the **Linker** folder.
3. Click the **System** property page.
4. Modify the **Driver** property.

To set this linker option programmatically

- See VCLinkerTool.driver Property.

See also

MSVC linker reference
MSVC Linker Options

`/DYNAMICBASE` (Use address space layout randomization)

Article • 05/06/2022

Specifies whether to generate an executable image that can be randomly rebased at load time by using the address space layout randomization (ASLR) feature of Windows. ASLR was first available in Windows Vista.

Syntax

> `/DYNAMICBASE`[`:NO`]

Remarks

The `/DYNAMICBASE` option modifies the header of an *executable image*, a .dll or .exe file, to indicate whether the application should be randomly rebased at load time, and enables virtual address allocation randomization, which affects the virtual memory location of heaps, stacks, and other operating system allocations. The `/DYNAMICBASE` option applies to both 32-bit and 64-bit images. ASLR is supported on Windows Vista and later operating systems. The option is ignored by earlier operating systems.

By default, `/DYNAMICBASE` is enabled. To disable this option, use `/DYNAMICBASE:NO`. The `/DYNAMICBASE` option is required for the /HIGHENTROPYVA option to have an effect.

Because ASLR can't be disabled on ARM, ARM64, or ARM64EC architectures, `/DYNAMICBASE:NO` isn't supported for these targets.

To set this linker option in Visual Studio

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Randomized Base Address** property.

To set this linker option programmatically

- See RandomizedBaseAddress.

See also

- MSVC linker reference
- MSVC linker options
- /HIGHENTROPYVA
- Windows ISV Software Security Defenses

/ENTRY (Entry-Point Symbol)

Article • 08/03/2021

```
/ENTRY:function
```

Arguments

function
A function that specifies a user-defined starting address for an .exe file or DLL.

Remarks

The /ENTRY option specifies an entry point function as the starting address for an .exe file or DLL.

The function must be defined to use the `__stdcall` calling convention. The parameters and return value depend on if the program is a console application, a windows application or a DLL. It is recommended that you let the linker set the entry point so that the C run-time library is initialized correctly, and C++ constructors for static objects are executed.

By default, the starting address is a function name from the C run-time library. The linker selects it according to the attributes of the program, as shown in the following table.

Function name	Default for
mainCRTStartup (or **wmainCRTStartup**)	An application that uses /SUBSYSTEM:CONSOLE; calls `main` (or `wmain`)
WinMainCRTStartup (or **wWinMainCRTStartup**)	An application that uses /SUBSYSTEM:**WINDOWS**; calls `WinMain` (or `wWinMain`), which must be defined to use `__stdcall`
_DllMainCRTStartup	A DLL; calls `DllMain` if it exists, which must be defined to use `__stdcall`

If the /DLL or /SUBSYSTEM option is not specified, the linker selects a subsystem and entry point depending on whether `main` or `WinMain` is defined.

The functions `main`, `WinMain`, and `DllMain` are the three forms of the user-defined entry point.

When creating a managed image, the function specified to /ENTRY must have a signature of (LPVOID *var1*, DWORD *var2*, LPVOID *var3*).

For information on how to define your own `DllMain` entry point, see DLLs and Visual C++ run-time library behavior .

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Entry Point** property.

To set this linker option programmatically

- See EntryPointSymbol.

See also

MSVC linker reference
MSVC Linker Options

/ERRORREPORT (Report Internal Linker Errors)

Article • 08/03/2021

The **/ERRORREPORT** option is deprecated. Starting in Windows Vista, error reporting is controlled by Windows Error Reporting (WER) settings.

Syntax

> **/ERRORREPORT:** [**none** | **prompt** | **queue** | **send**]

Remarks

The **/ERRORREPORT** arguments are overridden by the Windows Error Reporting service settings. The linker automatically sends reports of internal errors to Microsoft, if reporting is enabled by Windows Error Reporting. No report is sent if disabled by Windows Error Reporting.

To set this compiler option in the Visual Studio development environment

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Error Reporting** property.

To set this compiler option programmatically

- See ErrorReporting.

See also

MSVC linker reference
MSVC linker options

/EXPORT (Exports a Function)

Article • 08/03/2021

Exports a function by name or ordinal, or data, from your program.

Syntax

> **/EXPORT:***entryname*[,@*ordinal*[,**NONAME**]][,**DATA**]

Remarks

The **/EXPORT** option specifies a function or data item to export from your program so that other programs can call the function or use the data. Exports are usually defined in a DLL.

The *entryname* is the name of the function or data item as it is to be used by the calling program. *ordinal* specifies an index into the exports table in the range 1 through 65,535; if you do not specify *ordinal*, LINK assigns one. The **NONAME** keyword exports the function only as an ordinal, without an *entryname*.

The **DATA** keyword specifies that the exported item is a data item. The data item in the client program must be declared using **extern __declspec(dllimport)**.

There are four methods for exporting a definition, listed in recommended order of use:

1. __declspec(dllexport) in the source code
2. An EXPORTS statement in a .def file
3. An /EXPORT specification in a LINK command
4. A comment directive in the source code, of the form `#pragma comment(linker, "/export: definition ")`.

All these methods can be used in the same program. When LINK builds a program that contains exports, it also creates an import library, unless an .exp file is used in the build.

LINK uses decorated forms of identifiers. The compiler decorates an identifier when it creates the .obj file. If *entryname* is specified to the linker in its undecorated form (as it appears in the source code), LINK attempts to match the name. If it cannot find a unique match, LINK issues an error message. Use the DUMPBIN tool to get the decorated name form of an identifier when you need to specify it to the linker.

> **Note**
>
> Do not specify the decorated form of C identifiers that are declared `__cdecl` or `__stdcall`.

If you need to export an undecorated function name, and have different exports depending on the build configuration (for example, in 32-bit or 64-bit builds), you can use different DEF files for each configuration. (Preprocessor conditional directives are not allowed in DEF files.) As an alternative, you can use a `#pragma comment` directive before a function declaration as shown here, where `PlainFuncName` is the undecorated name, and `_PlainFuncName@4` is the decorated name of the function:

```
C++

#pragma comment(linker, "/export:PlainFuncName=_PlainFuncName@4")
BOOL CALLBACK PlainFuncName( Things * lpParams)
```

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC Linker Options

/FILEALIGN (Align sections in files)

Article • 08/03/2021

The **/FILEALIGN** linker option lets you specify the alignment of sections written to your output file as a multiple of an specified size.

Syntax

> **/FILEALIGN:***size*

Parameters

size
The section alignment size in bytes, which must be a power of two.

Remarks

The **/FILEALIGN** option causes the linker to align each section in the output file on a boundary that is a multiple of the *size* value. By default, the linker does not use a fixed alignment size.

The **/FILEALIGN** option can be used to make disk utilization more efficient, or to make page loads from disk faster. A smaller section size may be useful for apps that run on smaller devices, or to keep downloads smaller. Section alignment on disk does not affect alignment in memory.

Use DUMPBIN to see information about sections in your output file.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter the option name **/FILEALIGN:** and the size in the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC Linker Options

/FIXED (Fixed Base Address)

Article • 08/03/2021

```
/FIXED[:NO]
```

Remarks

Tells the operating system to load the program only at its preferred base address. If the preferred base address is unavailable, the operating system does not load the file. For more information, see /BASE (Base Address).

/FIXED:NO is the default setting for a DLL, and /FIXED is the default setting for any other project type.

When /FIXED is specified, LINK does not generate a relocation section in the program. At run time, if the operating system is unable to load the program at the specified address, it issues an error message and does not load the program.

Specify /FIXED:NO to generate a relocation section in the program.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter the option name and setting in the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC Linker Options

`/FORCE` (Force file output)

Article • 09/22/2022

Tells the linker to create an executable even if symbols are undefined or multiply defined.

Syntax

> `/FORCE`[`:MULTIPLE`|`:UNRESOLVED`]

Remarks

The `/FORCE` linker option tells the linker to create an executable image (EXE file or DLL) even if a symbol is referenced but not defined or is defined more than once.

> **ⓘ Important**
>
> The `/FORCE` option can create an executable that crashes or misbehaves at runtime if it references an undefined symbol or, when a multiply defined symbol has different definitions, if it invokes an unexpected definition in context.

The `/FORCE` option can take an optional argument:

- Use `/FORCE:MULTIPLE` to create an output file whether or not LINK finds more than one definition for a symbol.
- Use `/FORCE:UNRESOLVED` to create an output file whether or not LINK finds an undefined symbol. `/FORCE:UNRESOLVED` is ignored if the entry point symbol is unresolved.

`/FORCE` with no arguments implies both `/FORCE:MULTIPLE` and `/FORCE:UNRESOLVED`.

The linker won't link incrementally when the `/FORCE` option is specified.

If a module is compiled with `/clr`, the linker ignores the `/FORCE` option.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Force File Output** property. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options

`/FUNCTIONPADMIN` (Create hotpatchable image)

Article • 09/22/2022

Tells the linker to prepare an executable image for hot patching.

Syntax

> `/FUNCTIONPADMIN`[`:`*`size`]

Arguments

`size`
The amount of padding to add to the beginning of each function in bytes. On x86 the default is 5 bytes of padding and on x64 the default is 6 bytes. On other targets a value must be provided.

Remarks

In order for the linker to produce a hotpatchable image, the *`.obj`* files must be compiled by using the /hotpatch (Create hotpatchable image) compiler option.

When you compile and link an image with a single invocation of cl.exe, `/hotpatch` implies `/FUNCTIONPADMIN`.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Create Hot Patchable Image** property. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options

`/GENPROFILE`, `/FASTGENPROFILE` (Generate Profiling Instrumented Build)

Article • 08/03/2021

Specifies generation of a *`.pgd`* file by the linker to support profile-guided optimization (PGO). **`/GENPROFILE`** and **`/FASTGENPROFILE`** use different default parameters. Use **`/GENPROFILE`** to favor precision over speed and memory usage during profiling. Use **`/FASTGENPROFILE`** to favor smaller memory usage and speed over precision.

Syntax

> **`/GENPROFILE`**[**`:`***`profile-argument`*[**`,`***`profile-argument`** ...]]
> **`/FASTGENPROFILE`**[**`:`***`profile-argument`*[**`,`***`profile-argument`** ...]]\

> *`profile-argument`*
> { **`COUNTER32`** | **`COUNTER64`** }
> { **`EXACT`** | **`NOEXACT`** }
> **`MEMMAX=`***value*
> **`MEMMIN=`***value*
> { **`PATH`** | **`NOPATH`** }
> { **`TRACKEH`** | **`NOTRACKEH`** }
> **`PGD=`***filename*

Arguments

Any of the *`profile-argument`* arguments may be specified to **`/GENPROFILE`** or **`/FASTGENPROFILE`**. Arguments listed here separated by a pipe character (**`|`**) are mutually exclusive. Use a comma character (**`,`**) to separate arguments. Don't put spaces between arguments, commas, or after the colon (**`:`**).

`COUNTER32` | **`COUNTER64`**
Use **`COUNTER32`** to specify the use of 32-bit probe counters, and **`COUNTER64`** to specify 64-bit probe counters. When you specify **`/GENPROFILE`**, the default is **`COUNTER64`**. When you specify **`/FASTGENPROFILE`**, the default is **`COUNTER32`**.

`EXACT` | **`NOEXACT`**
Use **`EXACT`** to specify thread-safe interlocked increments for probes. **`NOEXACT`** specifies

unprotected increment operations for probes. The default is **NOEXACT**.

MEMMAX=*value*, **MEMMIN**=*value*
Use **MEMMAX** and **MEMMIN** to specify the maximum and minimum reservation sizes for training data in memory. The value is the amount of memory to reserve in bytes. By default, these values are determined by an internal heuristic.

PATH | **NOPATH**
Use **PATH** to specify a separate set of PGO counters for each unique path to a function. Use **NOPATH** to specify only one set of counters for each function. When you specify **/GENPROFILE**, the default is **PATH** . When you specify **/FASTGENPROFILE**, the default is **NOPATH** .

TRACKEH | **NOTRACKEH**
Specifies whether to use extra counters to keep an accurate count when exceptions are thrown during training. Use **TRACKEH** to specify extra counters for an exact count. Use **NOTRACKEH** to specify single counters for code that doesn't use exception handling or that doesn't run into exceptions in your training scenarios. When you specify **/GENPROFILE**, the default is **TRACKEH** . When you specify **/FASTGENPROFILE**, the default is **NOTRACKEH** .

PGD=*filename*
Specifies a base file name for the *.pgd* file. By default, the linker uses the base executable image file name with a *.pgd* extension.

Remarks

The **/GENPROFILE** and **/FASTGENPROFILE** options tell the linker to generate the profiling instrumentation file needed to support application training for profile-guided optimization (PGO). These options are new in Visual Studio 2015. Prefer these options to the deprecated **/LTCG:PGINSTRUMENT**, **/PGD**, and **/POGOSAFEMODE** options, and to the **PogoSafeMode**, **VCPROFILE_ALLOC_SCALE**, and **VCPROFILE_PATH** environment variables. The profiling information generated by application training is used as input for targeted whole-program optimizations during builds. You can also set other options to control various profiling features for performance during app training and builds. The default options specified by **/GENPROFILE** give the most accurate results, especially for large, complex multi-threaded apps. The **/FASTGENPROFILE** option uses different defaults for a lower memory footprint and faster performance during training, at the expense of accuracy.

Profiling information is captured when you run the instrumented app after you build by using `/GENPROFILE` of `/FASTGENPROFILE`. This information is captured when you specify the /USEPROFILE linker option to do the profiling step and then used to guide the optimized build step. For more information on how to train your app and details on the collected data, see Profile-guided optimizations.

Always specify `/LTCG` when you specify `/GENPROFILE` or `/FASTGENPROFILE`.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter the `/GENPROFILE` or `/FASTGENPROFILE` options and arguments into the **Additional Options** box. Choose `OK` to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options
/LTCG (Link-time code generation)

`/GUARD` (Enable Guard Checks)

Article • 09/22/2022

Tells the linker whether to support Control Flow Guard checks in the executable image.

Syntax

> `/GUARD:CF`
>
> `/GUARD:NO`

Remarks

The `/GUARD:CF` linker option modifies the header of a DLL or EXE file to indicate support for Control Flow Guard (CFG) runtime checks. The linker also adds the required control flow target address data to the header. By default, `/GUARD:CF` is disabled. It can be explicitly disabled by using `/GUARD:NO`. To be effective, `/GUARD:CF` also requires the /DYNAMICBASE (Use address space layout randomization) linker option, which is on by default.

When source code is compiled by using the /guard:cf compiler option, the compiler analyzes the control flow by examining all indirect calls for possible target addresses. The compiler inserts code to verify the target address of an indirect call instruction is in the list of known target addresses at runtime. Operating systems that support CFG stop a program that fails a CFG runtime check. This check makes it more difficult for an attacker to execute malicious code by using data corruption to change a call target.

The `/GUARD:CF` option must be specified to both the compiler and linker to create CFG-enabled executable images. Code compiled but not linked by using `/GUARD:CF` incurs the cost of runtime checks, but doesn't enable CFG protection. When the `/guard:cf` option is specified to the `cl` command to compile and link in one step, the compiler passes the flag to the linker. When the **Control Flow Guard** property is set in Visual Studio, the `/GUARD:CF` option is passed to both the compiler and linker. When object files or libraries have been compiled separately, the option must be explicitly specified in the `link` command.

To set this linker option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. In **Additional Options**, enter `/GUARD:CF`. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

/guard (Enable Control Flow Guard)
MSVC linker reference
MSVC linker options

/HEAP (Set Heap Size)

Article • 08/03/2021

```
/HEAP:reserve[,commit]
```

Remarks

The /HEAP option sets the size of the heap in bytes. This option is only for use when building an .exe file.

The *reserve* argument specifies the total heap allocation in virtual memory. The default heap size is 1 MB. The linker rounds up the specified value to the nearest 4 bytes.

The optional `commit` argument specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher `commit` value saves time when the application needs more heap space, but increases the memory requirements and possibly the startup time.

Specify the *reserve* and `commit` values in decimal or C-language notation.

This functionality is also available via a module definition file with HEAPSIZE.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **System** property page.
3. Modify the **Heap Commit Size** property.

To set this linker option programmatically

- See HeapReserveSize and HeapCommitSize.

See also

MSVC linker reference

MSVC Linker Options

/HIGHENTROPYVA (Support 64-Bit ASLR)

Article • 08/03/2021

Specifies whether the executable image supports high-entropy 64-bit address space layout randomization (ASLR).

Syntax

> `/HIGHENTROPYVA`[`:NO`]

Remarks

`/HIGHENTROPYVA` modifies the header of an *executable image* file (for example, a `.dll` or `.exe` file), to indicate whether ASLR can use the entire 64-bit address space. To have an effect, set the option on both the executable and all modules that it depends on. Then an operating system that supports 64-bit ASLR can rebase the executable image's segments at load time by using 64-bit randomized virtual addresses. This large address space makes it more difficult for an attacker to guess the location of a particular memory region.

By default, `/HIGHENTROPYVA` is enabled for 64-bit executable images. This option requires /LARGEADDRESSAWARE, which is also enabled by default for 64-bit images. `/HIGHENTROPYVA` isn't applicable to 32-bit executable images, where the linker ignores the option. To explicitly disable this option, use `/HIGHENTROPYVA:NO`.

For `/HIGHENTROPYVA` to have an effect at load time, /DYNAMICBASE must also be enabled. `/DYNAMICBASE` is enabled by default, and is required to enable ASLR in Windows Vista and later operating systems. Earlier versions of Windows ignore this flag.

To set this linker option in Visual Studio

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. In **Additional Options**, enter `/HIGHENTROPYVA` or `/HIGHENTROPYVA:NO`.

See also

- MSVC linker reference
- MSVC linker options
- /DYNAMICBASE
- /LARGEADDRESSAWARE
- Windows ISV Software Security Defenses

/IDLOUT (Name MIDL Output Files)

Article • 08/03/2021

```
/IDLOUT:[path\]filename
```

Parameters

path
An absolute or relative path specification. By specifying a path, you affect only the location of an .idl file; all other files are placed in the project directory.

filename
Specifies the name of the .idl file created by the MIDL compiler. No file extension is assumed; specify *filename*.idl if you want an .idl extension.

Remarks

The /IDLOUT option specifies the name and extension of the .idl file.

The MIDL compiler is called by the MSVC linker when linking projects that have the module attribute.

/IDLOUT also specifies the file names of the other output files associated with the MIDL compiler:

- *filename*.tlb
- *filename*_p.c
- *filename*_i.c
- *filename*.h

filename is the parameter that you pass to /IDLOUT. If /TLBOUT is specified, the .tlb file will get its name from /TLBOUT *filename*.

If you specify neither /IDLOUT nor /TLBOUT, the linker will create vc70.tlb, vc70.idl, vc70_p.c, vc70_i.c, and vc70.h.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Embedded IDL** property page.
3. Modify the **Merge IDL Base File Name** property.

To set this linker option programmatically

- See MergedIDLBaseFileName.

See also

MSVC linker reference
MSVC Linker Options
/IGNOREIDL (Don't Process Attributes into MIDL)
/MIDL (Specify MIDL Command Line Options)
Building an Attributed Program

/IGNORE (Ignore Specific Warnings)

Article • 08/03/2021

```
/IGNORE:warning[,warning]
```

Parameters

warning
The number of the linker warning to suppress, in the range 4000 to 4999.

Remarks

By default, LINK reports all warnings. Specify **/IGNORE:**`warning` to tell the linker to suppress a specific warning number. To ignore multiple warnings, separate the warning numbers with commas.

The linker does not allow some warnings to be ignored. This table lists the warnings that are not suppressed by **/IGNORE**:

Linker Warning	Message
LNK4017	`keyword` statement not supported for the target platform; ignored
LNK4044	unrecognized option '`option`'; ignored
LNK4062	'`option`' not compatible with '`architecture`' target machine; option ignored
LNK4075	ignoring "`option1`" due to "`option2`" specification
LNK4086	entrypoint '`function`' is not __stdcall with '`number`' bytes of arguments; image may not run
LNK4088	image being generated due to /FORCE option; image may not run
LNK4105	no argument specified with option '`option`'; ignoring switch
LNK4203	error reading program database '`filename`'; linking object as if no debug info
LNK4204	'`filename`' is missing debugging information for referencing module; linking object as if no debug info

Linker Warning	Message
LNK4205	'`filename`' is missing current debugging information for referencing module; linking object as if no debug info
LNK4206	precompiled type information not found; '`filename`' not linked or overwritten; linking object as if no debug info
LNK4207	'`filename`' compiled /Yc /Yu /Z7; cannot create PDB; recompile with /Zi; linking object as if no debug info
LNK4208	incompatible PDB format in '`filename`'; delete and rebuild; linking object as if no debug info
LNK4209	debugging information corrupt; recompile module; linking object as if no debug info
LNK4224	`option` is no longer supported; ignored
LNK4228	'`option`' invalid for a DLL; ignored
LNK4229	invalid directive /`directive` found; ignored

In general, linker warnings that can't be ignored represent build failures, command line errors or configuration errors that you should fix.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Modify the **Additional Options** property.

To set this linker option programmatically

- See AdditionalOptions.

/IGNOREIDL (Don't Process Attributes into MIDL)

Article • 11/11/2021

```
/IGNOREIDL
```

Remarks

The /IGNOREIDL option specifies that any IDL attributes in source code should not be processed into an .idl file.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Embedded IDL** property page.
3. Modify the **Ignore Embedded IDL** property.

To set this linker option programmatically

- See IgnoreEmbeddedIDL.

See also

MSVC linker reference
MSVC Linker Options
/IDLOUT (Name MIDL Output Files)
/TLBOUT (Name .TLB File)
/MIDL (Specify MIDL Command Line Options)
Building an Attributed Program

`/ILK` (Name incremental database file)

Article • 09/08/2022

The `/ILK` linker option tells the linker where to put the *`.ilk`* database file for incremental link information (/INCREMENTAL).

Syntax

> **`/ILK:`**[*`pathname`*]

Arguments

`pathname`
The destination directory and filename for the generated *`.ilk`* file. If the **`/ILK`** option isn't specified when **`/INCREMENTAL`** is used, the filename is created by appending *`.ilk`* to the target base filename.

Remarks

The `/ILK` linker option tells the linker the path and filename to use for the *`.ilk`* incremental database file when you specify /INCREMENTAL.

To set this compiler option in the Visual Studio development environment

1. Open the project **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Incremental Link Database File** property. The default value is `$(IntDir)$(TargetName).ilk`.

To set this compiler option programmatically

- See AdditionalOptions.

See also

/INCREMENTAL
MSVC linker reference
MSVC linker options

/IMPLIB (Name Import Library)

Article • 08/03/2021

> /IMPLIB:*filename*

Parameters

filename
A user-specified name for the import library. It replaces the default name.

Remarks

The /IMPLIB option overrides the default name for the import library that LINK creates when it builds a program that contains exports. The default name is formed from the base name of the main output file and the extension .lib. A program contains exports if one or more of the following are specified:

- The __declspec(dllexport) keyword in the source code
- EXPORTS statement in a .def file
- An /EXPORT specification in a LINK command

LINK ignores /IMPLIB when an import library is not being created. If no exports are specified, LINK does not create an import library. If an export file is used in the build, LINK assumes that an import library already exists and does not create one. For information on import libraries and export files, see LIB Reference.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Import Library** property.

To set this linker option programmatically

- See ImportLibrary.

See also

MSVC linker reference
MSVC Linker Options

/INCLUDE (Force Symbol References)

Article • 08/03/2021

```
/INCLUDE:symbol
```

Parameters

symbol
Specifies a symbol to be added to the symbol table.

Remarks

The /INCLUDE option tells the linker to add a specified symbol to the symbol table.

To specify multiple symbols, type a comma (,), a semicolon (;), or a space between the symbol names. On the command line, specify /INCLUDE:`symbol` once for each symbol.

The linker resolves `symbol` by adding the object that contains the symbol definition to the program. This feature is useful for including a library object that otherwise would not be linked to the program.

Specifying a symbol with this option overrides the removal of that symbol by /OPT:REF.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Input** property page.
3. Modify the **Force Symbol References** property.

To set this linker option programmatically

- See ForceSymbolReferences.

See also

MSVC linker reference
MSVC Linker Options

`/INCREMENTAL` (Link incrementally)

Article • 09/08/2022

Specifies whether to link incrementally or always perform a full link.

Syntax

> `/INCREMENTAL`[`:NO`]

Remarks

The `/INCREMENTAL` linker option controls how the linker handles incremental linking.

By default, the linker runs in incremental mode. To override a default incremental link, specify `/INCREMENTAL:NO`.

An incrementally linked program is functionally equivalent to a program that is non-incrementally linked. However, because it's prepared for subsequent incremental links, an incrementally linked executable, static library, or dynamic-link library file:

- Is larger than a non-incrementally linked program because of padding of code and data. Padding enables the linker to increase the size of functions and data without recreating the file.
- May contain jump thunks to handle relocation of functions to new addresses.

> **Note**
>
> To ensure that your final release build does not contain padding or thunks, link your program non-incrementally.

To link incrementally regardless of the default, specify `/INCREMENTAL`. When this option is selected, the linker issues a warning if it can't link incrementally, and then links the program non-incrementally. Certain options and situations override `/INCREMENTAL`.

Most programs can be linked incrementally. However, some changes are too great, and some options are incompatible with incremental linking. LINK performs a full link if any of the following options are specified:

- Link Incrementally isn't selected (`/INCREMENTAL:NO`)

- `/OPT:REF` is selected
- `/OPT:ICF` is selected
- `/OPT:LBR` is selected
- `/ORDER` is selected

`/INCREMENTAL` is implied when /DEBUG is specified.

Additionally, LINK performs a full link if any of the following situations occur:

- The incremental status (`.ilk`) file is missing. (LINK creates a new `.ilk` file in preparation for subsequent incremental linking.)
- There's no write permission for the `.ilk` file. (LINK ignores the `.ilk` file and links non-incrementally.)
- The `.exe` or `.dll` output file is missing.
- The timestamp of the `.ilk`, `.exe`, or `.dll` is changed.
- A LINK option is changed. Most LINK options, when changed between builds, cause a full link.
- An object (`.obj`) file is added or omitted.

An incremental link creates or updates an incremental link database `.ilk` file. You can specify the name and location of this file by using the /ILK (Name incremental database file) linker option. For more information about the `.ilk` file, see .ilk files as linker input.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Enable Incremental Linking** property.

To set this linker option programmatically

1. See LinkIncremental.

See also

MSVC linker reference
MSVC linker options
.ilk files as linker input

`/INFERASANLIBS` (Use inferred sanitizer libs)

Article • 08/03/2021

Use the `/INFERASANLIBS` linker option to enable or disable linking to the default AddressSanitizer libraries. As of Visual Studio 2019 16.9, the only supported sanitizer is AddressSanitizer.

Syntax

> `/INFERASANLIBS`[`:NO`]

Remarks

The `/INFERASANLIBS` linker option enables the default AddressSanitizer libraries. This option is enabled by default.

The `/INFERASANLIBS` and `/INFERASANLIBS:NO` linker options offer support for advanced users. For more information, see AddressSanitizer build and language reference.

The `/INFERASANLIBS` option is available beginning in Visual Studio 2019 version 16.9.

To set the `/INFERASANLIBS` linker option in the Visual Studio development environment

1. Open your project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Modify the **Additional Options** property. To enable default libraries, enter **/INFERASANLIBS** in the edit box. To disable default libraries, enter **/INFERASANLIBS:NO** instead.
4. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options
/fsanitize (Enable sanitizers)
AddressSanitizer overview
AddressSanitizer known issues
AddressSanitizer build and language reference

`/INTEGRITYCHECK` (Require signature check)

Article • 08/30/2023

Specifies that the digital signature of the binary image must be checked at load time.

> `/INTEGRITYCHECK`

Remarks

By default, `/INTEGRITYCHECK` is off.

The `/INTEGRITYCHECK` linker option sets a flag, `IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY`, in the PE header of the DLL file or executable file. This flag tells the memory manager to check for a digital signature in order to load the image in Windows. This option must be set for both 32-bit and 64-bit DLLs that certain Windows features load. It's recommended for all device drivers on Windows Vista, Windows Server 2008, and all later versions of Windows and Windows Server. Versions of Windows prior to Windows Vista ignore this flag. For more information, see Forced Integrity Signing of Portable Executable (PE) files.

Signing `/INTEGRITYCHECK` files

Microsoft has new signing guidance for DLL and executable files linked by using `/INTEGRITYCHECK`. The guidance used to recommend a cross-signed certificate from the cross-signing program. However, the cross-signing program is now deprecated. You must now sign your `/INTEGRITYCHECK` files by using the Microsoft Azure Code Signing program instead.

To set this linker option in Visual Studio

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. To create a digitally signed image, include `/INTEGRITYCHECK` in the **Additional Options** command line. A digitally signed image must pass a verification check before it's loaded. This feature is disabled by default.

4. Choose **OK** to save your changes.

See also

MSVC linker reference
MSVC linker options
Forced integrity signing of portable executable (PE) files
Kernel-mode code signing requirements
AppInit DLLs and Secure Boot

/KERNEL (Create kernel mode binary)

Article • 08/29/2023

Create a binary that is suitable for running in kernel mode.

Syntax

> `/KERNEL`

Remarks

Causes the linker to emit a warning if any object file or library linked in the binary wasn't compiled with /kernel.

Code that can run in kernel mode must be compiled with the `/kernel` option. If you link a binary that contains code that wasn't compiled with `/kernel`, the binary might not run correctly in kernel mode.

Code for kernel mode is compiled with a simplified set of C++ language features that are specific to code that runs in kernel mode. The compiler produces warnings for C++ language features that are potentially disruptive but can't be disabled. For more information about compiling code in kernel mode, see /kernel (Create kernel mode binary).

To set this linker option in Visual Studio

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. In **Additional Options**, enter `/KERNELMODE`.

See also

- MSVC linker reference
- MSVC linker options
- Compiler options: /kernel

/KEYCONTAINER (Specify a Key Container to Sign an Assembly)

Article • 03/03/2022

```
/KEYCONTAINER:name
```

Arguments

name
Container that contains the key. Place the string in double quotation marks (" ") if it contains a space.

Remarks

The linker creates a signed assembly by inserting a public key into the assembly manifest and signing the final assembly with the private key. To generate a key file, type sn -k *filename* at the command line. **sn -i** installs the key pair into a container.

If you compile with /LN, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly that includes an explicit reference to the module, via #using, or when linking with /ASSEMBLYMODULE.

You can also pass your encryption information to the compiler with /KEYFILE. Use /DELAYSIGN if you want a partially signed assembly. For more information on signing an assembly, see Strong Name Assemblies (Assembly Signing) (C++/CLI).

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC Linker Options

/KEYFILE (Specify Key or Key Pair to Sign an Assembly)

Article • 08/03/2021

```
/KEYFILE:filename
```

Arguments

filename
File that contains the key. Place the string in double quotation marks (" ") if it contains a space.

Remarks

The linker inserts the public key into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type sn -k *filename* at the command line. A signed assembly is said to have a strong name.

If you compile with /LN, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly that includes an explicit reference to the module, via #using, or when linking with /ASSEMBLYMODULE.

You can also pass your encryption information to the linker with /KEYCONTAINER. Use /DELAYSIGN if you want a partially signed assembly. For more information on signing an assembly, see Strong Name Assemblies (Assembly Signing) (C++/CLI) and Creating and Using Strong-Named Assemblies.

In case both **/KEYFILE** and **/KEYCONTAINER** are specified (either by command-line option or by custom attribute), the linker will first try the key container. If that succeeds, then the assembly is signed with the information in the key container. If the linker doesn't find the key container, it will try the file specified with /KEYFILE. If that succeeds, the assembly is signed with the information in the key file and the key information will be installed in the key container (similar to sn -i) so that on the next compilation, the key container will be valid.

A key file might contain only the public key.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /NOASSEMBLY

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC Linker Options

/LARGEADDRESSAWARE (Handle Large Addresses)

Article • 03/02/2024

```
/LARGEADDRESSAWARE[:NO]
```

Remarks

The /LARGEADDRESSAWARE option tells the linker that the application can handle addresses larger than 2 gigabytes. In the 64-bit compilers, this option is enabled by default. In the 32-bit compilers, `/LARGEADDRESSAWARE:NO` is enabled if `/LARGEADDRESSAWARE` is not otherwise specified on the linker line.

If an application was linked with `/LARGEADDRESSAWARE`, `DUMPBIN` /HEADERS will display information to that effect.

Linking 64-bit applications with **`/LARGEADDRESSAWARE:NO`** is not recommended because it restricts the available address space, which can result in runtime failures if the app exhausts memory. It may also prevent x64 apps from running on ARM64 systems because the emulation runtime will try to reserve 4GB of virtual address space. If the app was linked with `/LARGEADRESSAWARE:NO`, the app won't launch because it can't allocate that much address space.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **System** property page.
3. Modify the **Enable Large Addresses** property.

To set this linker option programmatically

- See LargeAddressAware.

See also

MSVC linker reference
MSVC Linker Options

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

/LIBPATH (Additional Libpath)

Article • 08/03/2021

```
/LIBPATH:dir
```

Parameters

dir
Specifies a path that the linker will search before it searches the path specified in the LIB environment option.

Remarks

Use the /LIBPATH option to override the environment library path. The linker will first search in the path specified by this option, and then search in the path specified in the LIB environment variable. You can specify only one directory for each /LIBPATH option you enter. If you want to specify more than one directory, you must specify multiple /LIBPATH options. The linker will then search the specified directories in order.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Additional Library Directories** property.

To set this linker option programmatically

- See AdditionalLibraryDirectories.

See also

MSVC linker reference
MSVC Linker Options

/LINKREPRO (Link repro directory name)

Article • 08/03/2021

Tells the linker or library tool to generate a link repro in a specified directory.

Syntax

> **/LINKREPRO:***directory-name*

Arguments

/LINKREPRO:*directory-name*
The user-specified directory to store the link repro in. Directory names that include spaces must be enclosed in double quotes.

Remarks

The **/LINKREPRO** option is used to create a *link repro*. It's a set of build artifacts that allow Microsoft to reproduce a problem that occurs at link time, or during library operations. It's useful for problems such as a backend crash involving Link-Time Code Generation (LTCG), an LNK1000 linker error, or a linker crash. The tool produces a link repro when you specify the **/LINKREPRO** linker option, or when you set the `link_repro` environment variable in your command-line build environment. For more information, see the Link repros section of How to report a problem with the Microsoft C++ toolset.

Both the **/LINKREPRO** linker option and the `link_repro` environment variable require you to specify an output directory for the link repro. On the command line or in the IDE, specify the directory by using a **/LINKREPRO:***directory-name* option. The *directory-name* you specify may be an absolute or relative path, but the directory must exist. The command-line option overrides any directory value set in the `link_repro` environment variable.

For information on how to limit link repro generation to a specific target file name, see the /LINKREPROTARGET option. This option can be used to specify a specific target to generate a link repro for. It's useful in complex builds that invoke the linker or library tool more than once.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.

3. Enter the **/LINKREPRO:***directory-name* option in the **Additional Options** box. The *directory-name* value you specify must exist. Choose **OK** to apply the change.

Once you've generated the link repro, open this property page again to remove the **/LINKREPRO** option from your builds.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options
/LINKREPROTARGET

`/LINKREPROFULLPATHRSP` (Generate file containing absolute paths of linked files)

Article • 06/13/2024

Generates a response file (.RSP) containing the absolute paths of all the files the linker took as input.

This flag was introduced in Visual Studio 2022 version 17.11.

Syntax

> `/LINKREPROFULLPATHRSP:filename`

Remarks

Rather than generating a full link repro like `/LINKREPRO` which copies all the files to a directory and creating a response file with relative paths to that directory, this option writes the names of the files used during linking to the specified file.

For example, given:

- a directory `c:\temp\test` that contains the files `test.cpp`, `f1.cpp`, `f2.cpp`
- the linker command line: `link f1.obj f2.obj test.obj /out:test.exe /LINKREPROFULLPATHRSP:test.rsp` The linker produces `test.rsp` containing the following lines to reflect the fully qualified paths of the input files:

Windows Command Prompt

```
"c:\temp\test\f1.obj"
"c:\temp\test\f2.obj"
"c:\temp\test\test.obj"
```

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Enter `/LINKREPROFULLPATHRSP:file.rsp` into **Additional Options**. Choose **OK** or **Apply** to apply the change.

To set this linker option programmatically

- See VCLinkerTool.AdditionalOptions

See also

MSVC linker reference
MSVC linker options

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

/LINKREPROTARGET (Link repro file name)

Article • 08/03/2021

Tells the linker or library tool to generate a link repro only when the target has the specified file name.

Syntax

> **/LINKREPROTARGET:***file-name*

Arguments

/LINKREPROTARGET:*file-name*
The target file name to filter on. A link repro is only generated when the named file is the output target. File names that include spaces must be enclosed in double quotes. The file name should include the base name and the extension, but not the path.

Remarks

The **/LINKREPROTARGET** option is used to specify a target file name to generate a *link repro* for. A link repro is a set of build artifacts that allow Microsoft to reproduce a problem that occurs at link time, or during library operations. The linker or library tool produces a link repro when you specify the /LINKREPRO option, or when you set the `link_repro` environment variable in your command-line build environment.

The **/LINKREPROTARGET** option is useful in complex builds that invoke the linker or library tool more than once. It lets you specify a specific target for the link repro, such as *problem.dll*. It lets you generate the link repro only when the tool produces a specific file.

For more information about how and when to create a link repro, see the Link repros section of How to report a problem with the Microsoft C++ toolset.

The **/LINKREPRO** and /OUT options must be set for the **/LINKREPROTARGET** option to have any effect.

/LINKREPROTARGET is available starting in Visual Studio 2019 version 16.1.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.

3. Enter the **/LINKREPROTARGET:***file-name* option in the **Additional Options** box. Choose **OK** to apply the change.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options
/LINKREPRO

`/LTCG` (Link-time code generation)

Article • 09/22/2022

Use `/LTCG` to perform whole-program optimization, or to create profile-guided optimization (PGO) instrumentation, perform training, and create profile-guided optimized builds.

Syntax

> `/LTCG`[`:`{`INCREMENTAL`|`NOSTATUS`|`STATUS`|`OFF`}]

These options are deprecated starting in Visual Studio 2015:

> `/LTCG:`{`PGINSTRUMENT`|`PGOPTIMIZE`|`PGUPDATE`}

Arguments

`INCREMENTAL`
(Optional) Specifies that the linker only applies whole program optimization or link-time code generation (LTCG) to files affected by an edit, instead of the entire project. By default, this flag isn't set when `/LTCG` is specified, and the entire project is linked by using whole program optimization.

`NOSTATUS` | `STATUS`
(Optional) Specifies whether the linker displays a progress indicator that shows what percentage of the link is complete. By default, this status information isn't displayed.

`OFF`
(Optional) Disables link-time code generation. The linker treats all modules compiled with `/GL` as if they're compiled without that option, and any MSIL modules cause the link to fail.

`PGINSTRUMENT`
(Optional) This option is deprecated starting in Visual Studio 2015. Instead, use `/LTCG` and /GENPROFILE or /FASTGENPROFILE to generate an instrumented build for profile-guided optimization. The data that is collected from instrumented runs is used to create an optimized image. For more information, see Profile-Guided Optimizations. The short form of this option is `/LTCG:PGI`.

`PGOPTIMIZE`

(Optional) This option is deprecated starting in Visual Studio 2015. Instead, use `/LTCG` and /USEPROFILE to build an optimized image. For more information, see Profile-Guided Optimizations. The short form of this option is `/LTCG:PGO`.

`PGUPDATE`

(Optional) This option is deprecated starting in Visual Studio 2015. Instead, use `/LTCG` and `/USEPROFILE` to rebuild an optimized image. For more information, see Profile-Guided Optimizations. The short form of this option is `/LTCG:PGU`.

Remarks

The `/LTCG` option tells the linker to call the compiler and perform whole-program optimization. You can also do profile guided optimization. For more information, see Profile-Guided Optimizations.

With the following exceptions, you can't add linker options to the PGO combination of `/LTCG` and `/USEPROFILE` that weren't specified in the previous PGO initialization combination of `/LTCG` and `/GENPROFILE` options:

- /BASE
- /FIXED
- `/LTCG`
- /MAP
- /MAPINFO
- /NOLOGO
- /OUT
- /PGD
- /PDB
- /PDBSTRIPPED
- /STUB
- /VERBOSE

Any linker options that are specified together with the `/LTCG` and `/GENPROFILE` options to initialize PGO don't have to be specified when you build by using `/LTCG` and `/USEPROFILE`; they're implied.

The rest of this article discusses the link-time code generation done by `/LTCG`.

`/LTCG` is implied with /GL.

The linker invokes link-time code generation if it's passed a module that was compiled by using `/GL` or an MSIL module (see .netmodule Files as Linker Input). If you don't explicitly specify `/LTCG` when you pass `/GL` or MSIL modules to the linker, the linker eventually detects this situation and restarts the link by using `/LTCG`. Explicitly specify `/LTCG` when you pass `/GL` and MSIL modules to the linker for the fastest possible build performance.

For even faster performance, use `/LTCG:INCREMENTAL`. This option tells the linker to reoptimize only the files affected by a source file change, instead of the entire project. This option can significantly reduce the link time required. This option isn't the same option as incremental linking. If you remove the `/LTCG:INCREMENTAL` option, also remove any /LTCGOUT option to improve build times and disk utilization.

`/LTCG` isn't valid for use with /INCREMENTAL.

When `/LTCG` is used to link modules compiled by using /Og, /O1, /O2, or /Ox, the following optimizations are performed:

- Cross-module inlining
- Interprocedural register allocation (64-bit operating systems only)
- Custom calling convention (x86 only)
- Small TLS displacement (x86 only)
- Stack double alignment (x86 only)
- Improved memory disambiguation (better interference information for global variables and input parameters)

> **Note**
>
> The linker determines which optimizations were used to compile each function and applies the same optimizations at link time.

Using `/LTCG` and `/O2` causes double-alignment optimization.

If `/LTCG` and `/O1` are specified, double alignment isn't performed. If most of the functions in an application are compiled for speed, with a few functions compiled for size (for example, by using the optimize pragma), the compiler double-aligns the functions that are optimized for size if they call functions that require double alignment.

If the compiler can identify all of the call sites of a function, the compiler ignores explicit calling-convention modifiers, and tries to optimize the function's calling convention:

- pass parameters in registers
- reorder parameters for alignment
- remove unused parameters

If a function is called through a function pointer, or if a function is called from outside a module that is compiled by using `/GL`, the compiler doesn't attempt to optimize the function's calling convention.

> ⓘ Note
>
> If you use `/LTCG` and redefine `mainCRTStartup`, your application can have unpredictable behavior that relates to user code that executes before global objects are initialized. There are three ways to address this issue: do not redefine `mainCRTStartup`, do not compile the file that contains `mainCRTStartup` by using `/LTCG`, or initialize global variables and objects statically.

`/LTCG` and MSIL Modules

Modules that are compiled by using /GL and /clr can be used as input to the linker when `/LTCG` is specified.

- `/LTCG` can accept native object files, and mixed native/managed object files (compiled by using `/clr`). The `/clr:pure` and `/clr:safe` compiler options are deprecated in Visual Studio 2015 and unsupported in Visual Studio 2017 and later.
- `/LTCG:PGI` doesn't accept native modules compiled by using `/GL` and `/clr`

To set this compiler option in the Visual Studio development environment

The Whole Program Optimization property sets several compiler and linker options, including `/LTCG`. We recommend you use this property to change the settings for an entire build configuration. To set Whole Program Optimization for your project:

1. Open the project **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **General** property page.
3. Modify the **Whole Program Optimization** property. Choose **OK** or **Apply** to save your changes.

You can also apply `/LTCG` to specific builds by choosing **Build** > **Profile Guided Optimization** on the menu bar, or by choosing one of the Profile Guided Optimization options on the shortcut menu for the project.

To enable Link Time Code Generation separately or set a specific Link Time Code Generation option:

1. Open the project **Property Pages** dialog box.
2. Select the **Configuration Properties** > **Linker** > **Optimization** property page.
3. Modify the **Link Time Code Generation** property to one of the following options:
 - **Default**
 - **Use Fast Link Time Code Generation (LTCG:incremental)**
 - **Use Link Time Code Generation (LTCG)**
 - **Profile Guided Optimization - Instrument (LTCG:PGInstrument)**
 - **Profile Guided Optimization - Optimization (LTCG:PGOptimize)**
 - **Profile Guided Optimization - Update (LTCG:PGUpdate)**
4. Choose **OK** or **Apply** to save your changes.

To specify whether the linker displays a progress indicator for Link Time Code Generation:

1. Open the project **Property Pages** dialog box.
2. Select the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Link Status** property. Choose **OK** or **Apply** to save your changes.

To set this compiler option programmatically

- See LinkTimeCodeGeneration.

See also

MSVC linker reference
MSVC linker options

`/LTCGOUT` (Name LTCG incremental object file)

Article • 09/02/2022

The `/LTCGOUT` linker option tells the linker where to put the intermediate `.iobj` object file for incremental link-time code generation (`/LTCG:INCREMENTAL`).

Syntax

> `/LTCGOUT:`[*pathname*]

Arguments

pathname
The optional destination directory and filename for the generated `.iobj` file. If the `/LTCGOUT` option isn't specified when `/LTCG:INCREMENTAL` is used, the filename is created by appending `.iobj` to the target base filename. If the `/LTCGOUT` option is specified with an empty *pathname* when `/LTCG:INCREMENTAL` isn't used, no `.iobj` file is generated.

Remarks

The `/LTCGOUT` linker option tells the linker the path and filename to use for the intermediate `.iobj` object file when you specify /LTCG:INCREMENTAL. If you remove the `/LTCG:INCREMENTAL` option from your project, you should also remove any `/LTCGOUT` option.

To set this compiler option in the Visual Studio development environment

1. Open the project **Property Pages** dialog box. For more information, see Set compiler and build properties.

2. Select the **Configuration Properties** > **Linker** > **Optimization** property page.

3. Modify the **Link Time Code Generation Object File** property. The option isn't set if this property is empty.

To set this compiler option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options

/MACHINE (Specify Target Platform)

Article • 03/28/2022

> `/MACHINE:{ARM|ARM64|ARM64EC|EBC|X64|X86}`

Remarks

The `/MACHINE` option specifies the target platform for the program.

Usually, you don't have to specify the `/MACHINE` option. LINK infers the machine type from the *`.obj`* files. However, in some circumstances, LINK cannot determine the machine type and issues a linker tools error LNK1113. If such an error occurs, specify `/MACHINE`.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Target Machine** property.

To set this linker option programmatically

1. See TargetMachine.

See also

MSVC linker reference
MSVC Linker Options

`/MANIFEST` (Create side-by-side assembly manifest)

Article • 09/22/2022

Specifies whether the linker should create a side-by-side manifest file.

Syntax

> `/MANIFEST`[`:`{`EMBED`[`,ID=`*`resource_id`*]|`NO`}]

Remarks

The `/MANIFEST` linker option tells the linker to create a side-by-side manifest file. For more information about manifest files, see Manifest files reference.

The default is `/MANIFEST`.

The `/MANIFEST:EMBED` option specifies that the linker should embed the manifest file in the image as a resource of type `RT_MANIFEST`. The optional `ID` parameter sets the resource ID to use for the manifest. Use a *`resource_id`* value of 1 for an executable file. Use a value of 2 for a DLL to enable it to specify private dependencies. If the `ID` parameter isn't specified, the default value is 2 if the `/DLL` option is set; otherwise, the default value is 1.

Beginning with Visual Studio 2008, manifest files for executables contain a section that specifies User Account Control (UAC) information. If you specify `/MANIFEST` but don't specify either /MANIFESTUAC or /DLL, a default UAC fragment that has the UAC level set to *`asInvoker`* is inserted into the manifest. For more information about UAC levels, see /MANIFESTUAC (Embeds UAC information in manifest).

To change the default behavior for UAC, set one of these options:

- Specify the `/MANIFESTUAC` option and set the UAC level to the desired value.
- Or, specify the `/MANIFESTUAC:NO` option if you don't want to generate a UAC fragment in the manifest.

If you don't specify `/MANIFEST` but do specify /MANIFESTDEPENDENCY attributes, a manifest file is created. A manifest file isn't created if you specify `/MANIFEST:NO`.

If you specify `/MANIFEST`, the name of the manifest file is the same as the full name of your output file, but with `.manifest` appended to the file name. For example, if your output file name is `MyFile.exe`, the manifest file name is `MyFile.exe.manifest`. If you specify `/MANIFESTFILE:name`, the name of the manifest is what you specify in `name`.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **Manifest File** property page.
3. Modify the **Generate Manifest** property. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

1. See GenerateManifest.

See also

Manifest files reference
/MANIFESTDEPENDENCY (Specify manifest dependencies)
/MANIFESTFILE (Name manifest file)
/MANIFESTUAC (Embeds UAC information in manifest)
MSVC linker reference
MSVC linker options

/MANIFESTDEPENDENCY (Specify Manifest Dependencies)

Article • 08/03/2021

```
/MANIFESTDEPENDENCY:manifest_dependency
```

Remarks

/MANIFESTDEPENDENCY lets you specify attributes that will be placed in the <dependency> section of the manifest file.

See /MANIFEST (Create Side-by-Side Assembly Manifest) for information on how to create a manifest file.

For more information on the <dependency> section of the manifest file, see Publisher Configuration Files.

/MANIFESTDEPENDENCY information can be passed to the linker in one of two ways:

- Directly on the command line (or in a response file) with /MANIFESTDEPENDENCY.
- Via the comment pragma.

The following example shows a /MANIFESTDEPENDENCY comment passed via pragma,

C++

```
#pragma comment(linker, "\"/manifestdependency:type='Win32'
name='Test.Research.SampleAssembly' version='6.0.0.0'
processorArchitecture='X86' publicKeyToken='0000000000000000'
language='*'\"")
```

which results in the following entry in the manifest file:

XML

```
<dependency>
  <dependentAssembly>
    <assemblyIdentity type='Win32' name='Test.Research.SampleAssembly'
version='6.0.0.0' processorArchitecture='X86'
publicKeyToken='0000000000000000' language='*' />
```

```
  </dependentAssembly>
</dependency>
```

The same /MANIFESTDEPENDENCY comments can be passed at the command line as follows:

Windows Command Prompt

```
"/manifestdependency:type='Win32' name='Test.Research.SampleAssembly'
version='6.0.0.0' processorArchitecture='X86'
publicKeyToken='0000000000000000' language='*'\"
```

The linker will collect /MANIFESTDEPENDENCY comments, eliminate duplicate entries, and then add the resulting XML string to the manifest file. If the linker finds conflicting entries, the manifest file will become corrupt and the application will fail to launch (an entry may be added to the event log, indicating the source of the failure).

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Manifest File** property page.
3. Modify the **Additional Manifest Dependencies** property.

To set this linker option programmatically

1. See AdditionalManifestDependencies.

See also

MSVC linker reference
MSVC Linker Options

/MANIFESTFILE (Name Manifest File)

Article • 08/03/2021

```
/MANIFESTFILE:filename
```

Remarks

/MANIFESTFILE lets you change the default name of the manifest file. The default name of the manifest file is the file name with .manifest appended.

/MANIFESTFILE will have no effect if you do not also link with /MANIFEST.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Manifest File** property page.
3. Modify the **Manifest File** property.

To set this linker option programmatically

1. See ManifestFile.

See also

MSVC linker reference
MSVC Linker Options

/MANIFESTINPUT (Specify Manifest Input)

Article • 09/22/2022

Specifies a manifest input file to include in the manifest that's embedded in the image.

Syntax

> `/MANIFESTINPUT:`*`filename`*

Parameters

`filename`
The manifest file to include in the embedded manifest.

Remarks

The `/MANIFESTINPUT` option specifies the path of an input file to use to create the embedded manifest in an executable image. If you have multiple manifest input files, use the switch multiple times: once for each input file. The manifest input files are merged to create the embedded manifest. This option requires the `/MANIFEST:EMBED` option.

This option can't be set directly in Visual Studio. Instead, use the **Additional Manifest Files** property of the project to specify additional manifest files to include. For more information, see Manifest Tool Property Pages.

See also

MSVC linker reference
MSVC Linker Options

/MANIFESTUAC (Embeds UAC information in manifest)

Article • 08/03/2021

Specifies whether User Account Control (UAC) information is embedded in the program manifest.

Syntax

> `/MANIFESTUAC`
>
> `/MANIFESTUAC:NO`
>
> `/MANIFESTUAC:` *`level`*
>
> `/MANIFESTUAC:` *`uiAccess`*
>
> `/MANIFESTUAC:` *`fragment`*

Parameters

`NO`
The linker doesn't embed UAC information in the program manifest.

`level`
`level=` followed by one of **`'asInvoker'`**, **`'highestAvailable'`**, or **`'requireAdministrator'`**. Defaults to **`'asInvoker'`**. For more information, see the Remarks section.

`uiAccess`
`uiAccess='true'` if you want the application to bypass user interface protection levels and drive input to higher-permission windows on the desktop; otherwise, **`uiAccess='false'`**. Defaults to **`uiAccess='false'`**. Set this argument to **`uiAccess='true'`** only for user interface accessibility applications.

`fragment`
A string that contains the *`level`* and *`uiAccess`* values. May optionally be enclosed in double quotes. For more information, see the Remarks section.

Remarks

If you specify multiple `/MANIFESTUAC` options on the command-line, the last one entered takes precedence.

The choices for `/MANIFESTUAC:`*`level`* are as follows:

- `level='asInvoker'`: The application runs at the same permission level as the process that started it. You can elevate the application to a higher permission level by selecting **Run as Administrator**.
- `level='highestAvailable'`: The application runs at the highest permission level that it can. If the user who starts the application is a member of the Administrators group, this option is the same as `level='requireAdministrator'`. If the highest available permission level is higher than the level of the opening process, the system prompts for credentials.
- `level='requireAdministrator'`: The application runs using administrator permissions. The user who starts the application must be a member of the Administrators group. If the opening process isn't running with administrative permissions, the system prompts for credentials.

You can specify both the *`level`* and *`uiAccess`* values in one step by using the `/MANIFESTUAC:`*`fragment`* option. The fragment must be in the following form:

> `/MANIFESTUAC:` [`"`] `level=` { `'asInvoker'` | `'highestAvailable'` | `'requireAdministrator'` } `uiAccess=` { `'true'` | `'false'` } [`"`]

For example:

```
/MANIFESTUAC:"level='highestAvailable' uiAccess='true'"
```

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Manifest File** property page.
3. Modify the **Enable User Account Control (UAC)**, **UAC Execution Level**, and **UAC Bypass UI Protection** properties.

To set this linker option programmatically

1. See EnableUAC, UACExecutionLevel, and UACUIAccess.

See also

MSVC linker reference
MSVC Linker Options

/MAP (Generate Mapfile)

Article • 08/03/2021

```
/MAP[:filename]
```

Arguments

filename
A user-specified name for the mapfile. It replaces the default name.

Remarks

The /MAP option tells the linker to create a mapfile.

By default, the linker names the mapfile with the base name of the program and the extension .map. The optional *filename* allows you to override the default name for a mapfile.

A mapfile is a text file that contains the following information about the program being linked:

- The module name, which is the base name of the file
- The timestamp from the program file header (not from the file system)
- A list of groups in the program, with each group's start address (as *section*:*offset*), length, group name, and class
- A list of public symbols, with each address (as *section*:*offset*), symbol name, flat address, and .obj file where the symbol is defined
- The entry point (as *section*:*offset*)

The /MAPINFO option specifies additional information to be included in the mapfile.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Debug** property page.
3. Modify the **Generate Map File** property.

To set this linker option programmatically

1. See GenerateMapFile and MapFileName.

See also

MSVC linker reference
MSVC Linker Options

/MAPINFO (Include Information in Mapfile)

Article • 08/03/2021

```
/MAPINFO:EXPORTS
```

Remarks

The /MAPINFO option tells the linker to include the specified information in a mapfile, which is created if you specify the /MAP option. EXPORTS tells the linker to include exported functions.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Debug** property page.
3. Modify of the **Map Exports** properties:

To set this linker option programmatically

- See MapExports.

See also

MSVC linker reference
MSVC Linker Options

/MERGE (Combine Sections)

Article • 08/03/2021

```
/MERGE:from=to
```

Remarks

The /MERGE option combines the first section (*from*) with the second section (*to*), naming the resulting section *to*. For example, `/merge:.rdata=.text`.

If the second section does not exist, LINK renames the section *from* as *to*.

The /MERGE option is useful for creating VxDs and overriding the compiler-generated section names.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Merge Sections** property.

To set this linker option programmatically

1. See MergeSections.

See also

MSVC linker reference
MSVC Linker Options

/MIDL (Specify MIDL Command Line Options)

Article • 08/03/2021

Specifies a response file for MIDL command line options

Syntax

> **/MIDL:@***file*

Arguments

file
The name of the file that contains MIDL command line options.

Remarks

All options for the conversion of an IDL file to a TLB file must be given in *file*; MIDL command-line options cannot be specified on the linker's command line. If /MIDL is not specified, the MIDL compiler will be invoked with only the IDL file name and no other options.

The file should contain one MIDL command-line option per line.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Embedded IDL** property page.
3. Modify the **MIDL Commands** property.

To set this linker option programmatically

- See MidlCommandFile.

See also

MSVC linker reference
MSVC Linker Options
/IDLOUT (Name MIDL Output Files)
/IGNOREIDL (Don't Process Attributes into MIDL)
/TLBOUT (Name .TLB File)
Building an Attributed Program

`/NATVIS` (Add Natvis to PDB)

Article • 09/22/2022

Specifies a debugger visualization file (a Natvis file) to embed in the PDB file generated by the linker.

Syntax

> `/NATVIS:`*`filename`*

Parameters

`filename`
The pathname for a Natvis file to add to the PDB file. It embeds the debugger visualizations in the Natvis file into the PDB.

Remarks

The `/NATVIS` linker option embeds the debugger visualizations defined in the Natvis file *`filename`* into the PDB file generated by LINK. A Natvis file has a *`.natvis`* extension. Embedding the information allows the debugger to display the visualizations independently of the Natvis file. You can use multiple `/NATVIS` options to embed more than one Natvis file in the generated PDB file. For more information on how to create and use Natvis files, see Create custom views of native objects in the Visual Studio debugger.

LINK ignores `/NATVIS` when a PDB file isn't created by using a /DEBUG option.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Add the *`/NATVIS`* option to the **Additional Options** text box. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options

/NOASSEMBLY (Create a MSIL Module)

Article • 08/03/2021

```
/NOASSEMBLY
```

Remarks

The /NOASSEMBLY option tells the linker to create an image for the current output file without a .NET Framework assembly. An MSIL output file without an assembly manifest is called a module.

By default, an assembly is created. You can also use the /LN (Create MSIL Module) compiler option to create a module.

Other linker options that affect assembly generation are:

- /ASSEMBLYDEBUG
- /ASSEMBLYLINKRESOURCE
- /ASSEMBLYMODULE
- /ASSEMBLYRESOURCE
- /DELAYSIGN
- /KEYFILE
- /KEYCONTAINER

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Turn Off Assembly Generation** property.

To set this linker option programmatically

- See TurnOffAssemblyGeneration.

See also

MSVC linker reference
MSVC Linker Options

`/NODEFAULTLIB` (Ignore Libraries)

Article • 09/22/2022

The `/NODEFAULTLIB` linker option tells the linker to remove one or more default libraries from the list of libraries it searches when it resolves external references.

Syntax

> `/NODEFAULTLIB`[`:`*`library`*]

Arguments

`library`
An optional library name that you want the linker to ignore when it resolves external references.

Remarks

To create an *`.obj`* file that contains no references to default libraries, use /Zl (Omit default library name).

By default, `/NODEFAULTLIB` removes all default libraries from the list of libraries it searches when resolving external references. The optional *`library`* parameter lets you remove a specified library from the list of libraries it searches when resolving external references. Specify one `/NODEFAULTLIB` option for each library you want to exclude.

The linker resolves references to external definitions by searching first in libraries that you explicitly specify, then in default libraries specified by the /DEFAULTLIB option, and then in default libraries named in *`.obj`* files.

`/NODEFAULTLIB:`*`library`* overrides `/DEFAULTLIB:`*`library`* when the same *`library`* name is specified in both.

If you use `/NODEFAULTLIB` to build your program without the C run-time library, you may also have to use the /ENTRY option to specify the entry-point function in your program. For more information, see CRT library features.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Select the **Configuration Properties** > **Linker** > **Input** property page.
3. Modify the **Ignore All Default Libraries** property. Or, specify a semicolon-separated list of the libraries you want to ignore in the **Ignore Specific Default Libraries** property. The **Linker** > **Command Line** property page shows the effect of the changes you make to these properties.
4. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See IgnoreDefaultLibraryNames and IgnoreAllDefaultLibraries.

See also

MSVC linker reference
MSVC linker options

/NOENTRY (No Entry Point)

Article • 08/03/2021

```
/NOENTRY
```

Remarks

The /NOENTRY option is required for creating a resource-only DLL that contains no executable code. For more information, see Creating a Resource-Only DLL.

Use this option to prevent LINK from linking a reference to `_main` into the DLL.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **No Entry Point** property.

To set this linker option programmatically

1. See ResourceOnlyDLL.

See also

Creating a Resource-Only DLL
MSVC linker reference
MSVC Linker Options

/NOFUNCTIONPADSECTION (Disable function padding)

Article • 01/10/2024

Disables function padding for functions in the specified section.

Syntax

```
/NOFUNCTIONPADSECTION:[name]
```

Arguments

name
The name of the section to disable x64 function padding in.

Remarks

You can instruct the linker to put a specified minimum number of bytes between functions with /FUNCTIONPADMIN (Create hotpatchable image) and /ARM64XFUNCTIONPADMINX64. This flag disables adding that padding for the specified sections.

To exclude multiple sections, specify the switch multiple times.

This flag is available starting with in Visual Studio 17.8 and later.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Modify the **Additional Options** property to include **/NOFUNCTIONPADSECTION:**`name`, where `name` is the name of the section to disable x64 function padding in, and then choose **OK**.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC Linker Options
MSVC linker reference

/NOLOGO (Suppress Startup Banner) (Linker)

Article • 08/03/2021

```
/NOLOGO
```

Remarks

The /NOLOGO option prevents display of the copyright message and version number.

This option also suppresses echoing of command files. For details, see LINK Command Files.

By default, this information is sent by the linker to the Output window. On the command line, it is sent to standard output and can be redirected to a file.

To set this linker option in the Visual Studio development environment

1. This option should only be used from the command line.

To set this linker option programmatically

1. This linker option cannot be changed programmatically.

See also

MSVC linker reference
MSVC Linker Options

`/NXCOMPAT` (Compatible with Data Execution Prevention)

Article • 09/22/2022

Indicates that an executable is compatible with the Windows Data Execution Prevention feature.

Syntax

> `/NXCOMPAT`[`:NO`]

Remarks

By default, `/NXCOMPAT` is on.

`/NXCOMPAT:NO` can be used to explicitly specify an executable as incompatible with Data Execution Prevention.

For more information about Data Execution Prevention, see these articles:

- Data Execution Prevention
- Data Execution Prevention (Windows Embedded)

To set this linker option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set compiler and build properties.
2. Choose the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Data Execution Prevention (DEP)** property. Choose **OK** or **Apply** to apply the change.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference

MSVC linker options

/OPT (Optimizations)

Article • 02/17/2022

Controls the optimizations that LINK performs during a build.

Syntax

> **/OPT:**{**REF** | **NOREF**}
> **/OPT:**{**ICF**[=*iterations*] | **NOICF**}
> **/OPT:**{**LBR** | **NOLBR**}

Arguments

REF | **NOREF**

/OPT:REF eliminates functions and data that are never referenced; **/OPT:NOREF** keeps functions and data that are never referenced.

When /OPT:REF is enabled, LINK removes unreferenced packaged functions and data, known as *COMDATs*. This optimization is known as transitive COMDAT elimination. The **/OPT:REF** option also disables incremental linking.

Inlined functions and member functions defined inside a class declaration are always COMDATs. All of the functions in an object file are made into COMDATs if it is compiled by using the /Gy option. To place `const` data in COMDATs, you must declare it by using `__declspec(selectany)`. For information about how to specify data for removal or folding, see selectany.

By default, **/OPT:REF** is enabled by the linker unless **/OPT:NOREF** or /DEBUG is specified. To override this default and keep unreferenced COMDATs in the program, specify **/OPT:NOREF**. You can use the /INCLUDE option to override the removal of a specific symbol.

If /DEBUG is specified, the default for **/OPT** is **NOREF**, and all functions are preserved in the image. To override this default and optimize a debug build, specify **/OPT:REF**. This can reduce the size of your executable, and can be a useful optimization even in debug builds. We recommend that you also specify **/OPT:NOICF** to preserve identical functions in debug builds. This makes it easier to read stack traces and set breakpoints in functions that would otherwise be folded together.

ICF[=*iterations*] | **NOICF**

Use **ICF**[=*iterations*] to perform identical COMDAT folding. Redundant COMDATs can be removed from the linker output. The optional *iterations* parameter specifies the number of times to traverse the symbols for duplicates. The default number of iterations is 1. Additional iterations may locate more duplicates that are uncovered through folding in the previous iteration.

By default, **/OPT:ICF** is enabled by the linker unless **/OPT:NOICF** or /DEBUG is specified. To override this default and prevent COMDATs from being folded in the program, specify **/OPT:NOICF**.

In a debug build, you must explicitly specify **/OPT:ICF** to enable COMDAT folding. However, because **/OPT:ICF** can merge identical data or functions, it can change the function names that appear in stack traces. It can also make it impossible to set breakpoints in certain functions or to examine some data in the debugger, and can take you into unexpected functions when you single-step through your code. The behavior of the code is identical, but the debugger presentation can be very confusing. Therefore, we do not recommend that you use **/OPT:ICF** in debug builds unless the advantages of smaller code outweigh these disadvantages.

> ⓘ **Note**
>
> Because **/OPT:ICF** can cause the same address to be assigned to different functions or read-only data members (that is, `const` variables when compiled by using **/Gy**), it can break a program that depends on unique addresses for functions or read-only data members. For more information, see **/Gy (Enable Function-Level Linking)**.

LBR | NOLBR

The **/OPT:LBR** and **/OPT:NOLBR** options apply only to ARM binaries. Because certain ARM processor branch instructions have a limited range, if the linker detects a jump to an out-of-range address, it replaces the branch instruction's destination address with the address of a code "island" that contains a branch instruction that targets the actual destination. You can use **/OPT:LBR** to optimize the detection of long branch instructions and the placement of intermediate code islands to minimize overall code size. **/OPT:NOLBR** instructs the linker to generate code islands for long branch instructions as they are encountered, without optimization.

By default, the **/OPT:LBR** option is set when incremental linking is not enabled. If you want a non-incremental link but not long branch optimizations, specify **/OPT:NOLBR**. The **/OPT:LBR** option disables incremental linking.

Remarks

When used at the command line, the linker defaults to **/OPT:REF,ICF,LBR**. If **/DEBUG** is specified, the default is **/OPT:NOREF,NOICF,NOLBR**.

The **/OPT** optimizations generally decrease the image size and increase the program speed. These improvements can be substantial in larger programs, which is why they are enabled by default for retail builds.

Linker optimization does take extra time up front, but the optimized code also saves time when the linker has fewer relocations to fix up and creates a smaller final image, and it saves even more time when it has less debug information to process and write into the PDB. When optimization is enabled, it can result in a faster link time overall, as the small additional cost in analysis may be more than offset by the time savings in linker passes over smaller binaries.

The **/OPT** arguments may be specified together, separated by commas. For example, instead of **/OPT:REF /OPT:NOICF**, you can specify **/OPT:REF,NOICF**.

You can use the /VERBOSE linker option to see the functions that are removed by **/OPT:REF** and the functions that are folded by **/OPT:ICF**.

The **/OPT** arguments are often set for projects created by using the **New Project** dialog in the Visual Studio IDE, and usually have different values for debug and release configurations. If no value is set for these linker options in your project, then you may get the project defaults, which can be different from the default values used by the linker at the command line.

To set the OPT:ICF or OPT:REF linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Optimization** property page.
3. Modify one of these properties:
 - **Enable COMDAT Folding**
 - **References**

To set the OPT:LBR linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.

3. Enter the option in **Additional Options**:

 `/opt:lbr` or `/opt:nolbr`

To set this linker option programmatically

- See EnableCOMDATFolding and OptimizeReferences properties.

See also

- MSVC linker reference
- MSVC Linker Options

/ORDER (Put Functions in Order)

Article • 08/03/2021

Specify the link order for separately packaged (COMDAT) functions.

Syntax

> **/ORDER:@***filename*

Parameters

filename
A text file that specifies the link order for COMDAT functions.

Remarks

The **/ORDER** compiler option allows you to optimize your program's paging behavior by grouping a function together with the functions it calls. You can also group frequently called functions together. These techniques, known as *swap tuning* or *paging optimization*, increase the probability that a called function is in memory when it is needed and does not have to be paged from disk.

When you compile your source code into an object file, you can tell the compiler to put each function into its own section, called a *COMDAT*, by using the /Gy (Enable function-level linking) compiler option. The **/ORDER** linker option tells the linker to place COMDATs into the executable image in the order you specify.

To specify the COMDAT order, create a *response file*, a text file that lists each COMDAT by name, one per line, in the order you want them to be placed by the linker. Pass the name of this file as the *filename* parameter of the **/ORDER** option. For C++ functions, the name of a COMDAT is the decorated form of the function name. Use the undecorated name for C functions, `main`, and for C++ functions declared as `extern "C"`. Function names and decorated names are case sensitive. For more information on decorated names, see Decorated Names.

To find the decorated names of your COMDATs, use the DUMPBIN tool's /SYMBOLS option on the object file. The linker automatically prepends an underscore (_) to function names in the response file unless the name starts with a question mark (**?**) or at sign (**@**). For example, if a source file, example.cpp, contains functions `int cpp_func(int)`, `extern`

`"C" int c_func(int)` and `int main(void)`, the command `DUMPBIN /SYMBOLS example.obj` lists these decorated names:

Output

```
...
088 00000000 SECT1A notype ()    External     | ?cpp_func@@YAHH@Z (int
__cdecl cpp_func(int))
089 00000000 SECT22 notype ()    External     | _c_func
08A 00000000 SECT24 notype ()    External     | _main
...
```

In this case, specify the names as `?cpp_func@@YAHH@Z`, `c_func`, and `main` in your response file.

If more than one **/ORDER** option appears in the linker options, the last one specified takes effect.

The **/ORDER** option disables incremental linking. You may see linker warning LNK4075 when you specify this option if incremental linking is enabled, or if you have specified the /ZI (Incremental PDB) compiler option. To silence this warning, you can use the /INCREMENTAL:NO linker option to turn off incremental linking, and use the /Zi (Generate PDB) compiler option to generate a PDB without incremental linking.

> **Note**
>
> LINK cannot order static functions because static function names are not public symbol names. When **/ORDER** is specified, linker warning **LNK4037** is generated for each symbol in the order response file that is either static or not found.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Optimization** property page.
3. Modify the **Function Order** property to contain the name of your response file.

To set this linker option programmatically

- See FunctionOrder.

See also

MSVC linker reference
MSVC Linker Options

/OUT (Output File Name)

Article • 08/03/2021

```
/OUT:filename
```

Arguments

filename
A user-specified name for the output file. It replaces the default name.

Remarks

The /OUT option overrides the default name and location of the program that the linker creates.

By default, the linker forms the file name using the base name of the first .obj file specified and the appropriate extension (.exe or .dll).

This option the default base name for a .mapfile or import library. For details, see Generate Mapfile (/MAP) and /IMPLIB.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Output File** property.

To set this linker option programmatically

- See OutputFile.

See also

MSVC linker reference

MSVC Linker Options

/PDB (Use Program Database)

Article • 08/03/2021

```
/PDB:filename
```

Arguments

filename
A user-specified name for the program database (PDB) that the linker creates. It replaces the default name.

Remarks

By default, when /DEBUG is specified, the linker creates a program database (PDB) which holds debugging information. The default file name for the PDB has the base name of the program and the extension .pdb.

Use /PDB:*filename* to specify the name of the PDB file. If /DEBUG is not specified, the /PDB option is ignored.

A PDB file can be up to 2GB.

For more information, see .pdb Files as Linker Input.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Debug** property page.
3. Modify the **Generate Program Database File** property.

To set this linker option programmatically

- See ProgramDatabaseFile.

See also

MSVC linker reference
MSVC Linker Options

/PDBALTPATH (Use Alternate PDB Path)

Article • 08/03/2021

```
/PDBALTPATH:pdb_file_name
```

Arguments

pdb_file_name
The path and file name for the .pdb file.

Remarks

Use this option to provide an alternate location for the Program Database (.pdb) file in a compiled binary file. Normally, the linker records the location of the .pdb file in the binaries that it produces. You can use this option to provide a different path and file name for the .pdb file. The information provided with /PDBALTPATH does not change the location or name of the actual .pdb file; it changes the information that the linker writes in the binary file. This enables you to provide a path that is independent of the file structure of the build computer. Two common uses for this option are to provide a network path or a file that has no path information.

The value of *pdb_file_name* can be an arbitrary string, an environment variable, or %**_PDB**%. The linker will expand an environment variable, such as %**SystemRoot**%, to its value. The linker defines the environment variables %**_PDB**% and %**_EXT**%. %**_PDB**% expands to the file name of the actual .pdb file without any path information and %**_EXT**% is the extension of the generated executable.

See also

DUMPBIN Options
/PDBPATH

/PDBSTRIPPED (Strip Private Symbols)

Article • 08/03/2021

```
/PDBSTRIPPED:pdb_file_name
```

Arguments

pdb_file_name
A user-specified name for the stripped program database (PDB) that the linker creates.

Remarks

The /PDBSTRIPPED option creates a second program database (PDB) file when you build your program image with any of the compiler or linker options that generate a PDB file (/DEBUG, /Z7, /Zd, or /Zi). This second PDB file omits symbols that you would not want to ship to your customers. The second PDB file will only contain:

- Public symbols
- The list of object files and the portions of the executable to which they contribute
- Frame pointer optimization (FPO) debug records used to traverse the stack

The stripped PDB file will not contain:

- Type information
- Line number information
- Per-object file CodeView symbols such as those for functions, locals, and static data

The full PDB file will still be generated when you use /PDBSTRIPPED.

If you do not create a PDB file, /PDBSTRIPPED is ignored.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Debug** property page.
3. Modify the **Strip Private Symbols** property.

To set this linker option programmatically

- See StripPrivateSymbols.

See also

MSVC linker reference
MSVC Linker Options

/PGD (Specify Database for Profile-Guided Optimizations)

Article • 08/03/2021

The /PGD option is deprecated. Starting in Visual Studio 2015, prefer the /GENPROFILE or /FASTGENPROFILE linker options instead. This option is used to specify the name of the .pgd file used by the profile-guided optimization process.

Syntax

> **/PGD:***filename*

Argument

filename
Specifies the name of the .pgd file that is used to hold information about the running program.

Remarks

When using the deprecated /LTCG:PGINSTRUMENT option, use **/PGD** to specify a nondefault name or location for the .pgd file. If you do not specify **/PGD**, the .pgd file base name is the same as the output file (.exe or .dll) base name and is created in the same directory from which the link was invoked.

When using the deprecated **/LTCG:PGOPTIMIZE** option, use the **/PGD** option to specify the name of the .pgd file to use to create the optimized image. The *filename* argument should match the *filename* specified to **/LTCG:PGINSTRUMENT**.

For more information, see Profile-Guided Optimizations.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Linker** > **Optimization** property page.

3. Modify the **Profile Guided Database** property. Choose **OK** to save your changes.

To set this linker option programmatically

1. See ProfileGuidedDatabase.

See also

MSVC linker reference
MSVC Linker Options

/POGOSAFEMODE (Run PGO in thread safe mode)

Article • 08/03/2021

The /POGOSAFEMODE option is deprecated starting in Visual Studio 2015. Use the /GENPROFILE:EXACT and **/GENPROFILE:NOEXACT** options instead. The **/POGOSAFEMODE** linker option specifies that the instrumented build is created to use thread-safe mode for profile data capture during profile-guided optimization (PGO) training runs.

Syntax

> **/POGOSAFEMODE**

Remarks

Profile-guided optimization (PGO) has two possible modes during the profiling phase: *fast mode* and *safe mode*. When profiling is in fast mode, it uses an increment instruction to increase data counters. The increment instruction is faster but is not thread-safe. When profiling is in safe mode, it uses the interlocked-increment instruction to increase data counters. This instruction has the same functionality as the increment instruction has, and is thread-safe, but it is slower.

The **/POGOSAFEMODE** option sets the instrumented build to use safe mode. This option can only be used when the deprecated /LTCG:PGINSTRUMENT is specified, during the PGO instrumentation linker phase.

By default, PGO profiling operates in fast mode. **/POGOSAFEMODE** is only required if you want to use safe mode.

To run PGO profiling in safe mode, you must use either **/GENPROFILE:EXACT** (preferred), or use the environment variable PogoSafeMode or the linker switch **/POGOSAFEMODE**, depending on the system. If you are performing the profiling on an x64 computer, you must use the linker switch. If you are performing the profiling on an x86 computer, you may use the linker switch or define the environment variable to any value before you start the PGO instrumentation process.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Optimization** property page.
3. In the **Link Time Code Generation** property, choose **Profile Guided Optimization - Instrument (/LTCG:PGInstrument)**.
4. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
5. Enter the **/POGOSAFEMODE** option into the **Additional Options** box. Choose **OK** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

/GENPROFILE and /FASTGENPROFILE
/LTCG
Profile-Guided Optimizations
Environment Variables for Profile-Guided Optimizations

`/PROFILE` (Performance Tools Profiler)

Article • 10/14/2021

Produces an output file that can be used with the Performance Tools profiler.

Syntax

> `/PROFILE`

Remarks

/PROFILE implies the following linker options:

- /DEBUG:FULL
- /DEBUGTYPE:cv,fixup
- /OPT:REF
- /OPT:NOICF
- /INCREMENTAL:NO
- /FIXED:NO

`/PROFILE` is used to support the Performance Tools for Visual Studio Profiler utility VSInstr.exe.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.
3. Modify the **Profile** property.

To set this linker option programmatically

1. See Profile.

To set this linker option in a Visual Studio CMake project

Because a **CMake** project doesn't have the usual **Property Pages** support, the linker option can be set by modifying the *CMakeLists.txt* file.

1. Open the *CMakeLists.txt* file in the project root directory.
2. Add the code below. For more information, see the CMake set_target_properties documentation.

   ```txt
   SET_TARGET_PROPERTIES(${PROJECT_NAME} PROPERTIES LINK_FLAGS "/PROFILE")
   ```

3. Rebuild your solution.

See Also

MSVC linker reference
MSVC linker options

/RELEASE (Set the Checksum)

Article • 08/03/2021

```
/RELEASE
```

Remarks

The /RELEASE option sets the Checksum in the header of an .exe file.

The operating system requires the Checksum for device drivers. Set the Checksum for release versions of your device drivers to ensure compatibility with future operating systems.

The /RELEASE option is set by default when the /SUBSYSTEM:NATIVE option is specified.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Click the **Linker** folder.
3. Click the **Advanced** property page.
4. Modify the **Set Checksum** property.

To set this linker option programmatically

- See SetChecksum.

See also

MSVC linker reference
MSVC Linker Options

`/SAFESEH` (Image has Safe Exception Handlers)

Article • 09/22/2022

When `/SAFESEH` is specified, the linker only produces an image if it can also produce a table of the image's safe exception handlers. This table specifies to the operating system which exception handlers are valid for the image.

Syntax

> `/SAFESEH`
>
> `/SAFESEH:NO`

Remarks

`/SAFESEH` is only valid when linking for x86 targets. `/SAFESEH` isn't supported for platforms that already have the exception handlers noted. For example, on x64 and ARM, all exception handlers are noted in the PDATA. ML64.exe has support for adding annotations that emit SEH information (XDATA and PDATA) into the image, allowing you to unwind through ml64 functions. For more information, see MASM for x64 (ml64.exe).

If `/SAFESEH` isn't specified, the linker will produce an image with a table of safe exceptions handlers if all code segments are compatible with the safe exception handling feature. If any code segments weren't compatible with the safe exception handling feature, the resulting image won't contain a table of safe exception handlers. If /SUBSYSTEM specifies `WINDOWSCE` or one of the `EFI_*` options, the linker won't attempt to produce an image with a table of safe exceptions handlers, as neither of those subsystems can make use of the information.

If `/SAFESEH:NO` is specified, the linker won't produce an image with a table of safe exceptions handlers even if all code segments are compatible with the safe exception handling feature.

The most common reason the linker can't produce an image is because one or more of the input files to the linker was incompatible with the safe exception handlers feature. A common reason why code is incompatible with safe exception handlers is because it was created with a compiler from a previous version of Visual C++.

You can also register a function as a structured exception handler by using .SAFESEH.

It isn't possible to mark an existing binary as having safe exception handlers (or no exception handlers); information on safe exception handling must be added at build time.

The linker's ability to build a table of safe exception handlers depends on the application using the C runtime library. If you link with /NODEFAULTLIB and you want a table of safe exception handlers, you need to supply a load config struct (such as can be found in the `Loadcfg.c` CRT source file) that contains all the entries defined for Visual C++. For example:

C

```
#include <windows.h>
extern DWORD_PTR __security_cookie;  /* /GS security cookie */

/*
* The following two names are automatically created by the linker for any
* image that has the safe exception table present.
*/

extern PVOID __safe_se_handler_table[]; /* base of safe handler entry table
*/
extern BYTE  __safe_se_handler_count;  /* absolute symbol whose address is
                                          the count of table entries */
typedef struct {
    DWORD       Size;
    DWORD       TimeDateStamp;
    WORD        MajorVersion;
    WORD        MinorVersion;
    DWORD       GlobalFlagsClear;
    DWORD       GlobalFlagsSet;
    DWORD       CriticalSectionDefaultTimeout;
    DWORD       DeCommitFreeBlockThreshold;
    DWORD       DeCommitTotalFreeThreshold;
    DWORD       LockPrefixTable;            // VA
    DWORD       MaximumAllocationSize;
    DWORD       VirtualMemoryThreshold;
    DWORD       ProcessHeapFlags;
    DWORD       ProcessAffinityMask;
    WORD        CSDVersion;
    WORD        Reserved1;
    DWORD       EditList;                   // VA
    DWORD_PTR   *SecurityCookie;
    PVOID       *SEHandlerTable;
    DWORD       SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY32_2;

const IMAGE_LOAD_CONFIG_DIRECTORY32_2 _load_config_used = {
    sizeof(IMAGE_LOAD_CONFIG_DIRECTORY32_2),
    0,
    0,
    0,
```

```
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    &__security_cookie,
    __safe_se_handler_table,
    (DWORD)(DWORD_PTR) &__safe_se_handler_count
};
```

To set this linker option in the Visual Studio development environment

1. Open the **Property Pages** dialog box for the project. For more information, see Set compiler and build properties.

2. Select the **Configuration Properties** > **Linker** > **Advanced** property page.

3. Modify the **Image Has Safe Exception Handlers** property. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options

`/SECTION` (Specify Section Attributes)

Article • 09/22/2022

> **`/SECTION:`***`name`*`,`[[`!`]{`D`|`E`|`K`|`P`|`R`|`S`|`W`}][`,ALIGN=`*`number`*]

Remarks

The `/SECTION` option changes the attributes of a section, overriding the attributes set when the *`.obj`* file for the section was compiled.

A *section* in a portable executable (PE) file is a named contiguous block of memory that contains either code or data. Some sections contain code or data that your program declared and uses directly. Other data sections are created for you by the linker and library manager (LIB) and contain information vital to the operating system. For more information, see PE Format.

Specify a colon (`:`) and a section name *`name`*. The *`name`* is case sensitive.

Don't use the following names, as they conflict with standard names. For example, `.sdata` is used on RISC platforms:

- `.arch`
- `.bss`
- `.data`
- `.edata`
- `.idata`
- `.pdata`
- `.rdata`
- `.reloc`
- `.rsrc`
- `.sbss`
- `.sdata`

- `.srdata`
- `.text`
- `.xdata`

Specify one or more attributes for the section. The attribute characters, listed below, aren't case sensitive. You must specify all attributes that you want the section to have. An omitted attribute character causes that attribute bit to be turned off. If you don't specify `R`, `W`, or `E`, the existing read, write, or executable status remains unchanged.

To negate an attribute, precede its character with an exclamation point (`!`). The meanings of the attribute characters are shown in this table:

Character	Attribute	Meaning
`E`	Execute	The section is executable
`R`	Read	Allows read operations on data
`W`	Write	Allows write operations on data
`S`	Shared	Shares the section among all processes that load the image
`D`	Discardable	Marks the section as discardable
`K`	Cacheable	Marks the section as not cacheable
`P`	Pageable	Marks the section as not pageable

`K` and `P` are unusual in that the section flags that correspond to them are used in the negative sense. If you specify one of them on the `.text` section by using the **`/SECTION:.text,K`** option, there's no difference in the section flags when you run DUMPBIN with the /HEADERS option; the section was already implicitly cached. To remove the default, specify **`/SECTION:.text,!K`** instead. DUMPBIN reveals section characteristics, including "Not Cached."

A section in the PE file that doesn't have `E`, `R`, or `W` set is probably invalid.

The **`ALIGN=`***`number`* argument lets you specify an alignment value for a particular section. The *`number`* argument is in bytes and must be a power of two. For more information, see /ALIGN.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. Choose the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Specify Section Attributes** property. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options

/SOURCELINK (Include Source Link file in PDB)

Article • 08/03/2021

Specifies a Source Link configuration file to include in the PDB file generated by the linker.

Syntax

> `/SOURCELINK:`*`filename`*

Arguments

filename
Specifies a JSON-formatted configuration file that contains a simple mapping of local file paths to URLs for source files to display in the debugger. For more information on the format of this file, see Source Link JSON Schema.

Remarks

Source Link is a language- and source-control agnostic system for providing source debugging for binaries. Source Link is supported for native C++ binaries starting in Visual Studio 2017 version 15.8. For an overview of Source Link, see Source Link. For information on how to use Source Link in your projects, and how to generate the SourceLink file as part of your project, see Using Source Link.

To set the /SOURCELINK linker option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. In the **Additional options** box, add `/SOURCELINK:`*`filename`* and then choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- This option doesn't have a programmatic equivalent.

See also

MSVC linker reference
MSVC Linker Options

`/STACK` (Stack allocations)

Article • 02/23/2022

> `/STACK:` *`reserve`*[`,` *`commit`*]

Remarks

The `/STACK` linker option sets the size of the stack in bytes. Use this option only when you build an *`.exe`* file. The `/STACK` option is ignored when applied to *`.dll`* files.

The *`reserve`* value specifies the total stack allocation in virtual memory. For ARM64, x86, and x64 machines, the default stack size is 1 MB.

The *`commit`* value is subject to interpretation by the operating system. In WindowsRT, it specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher *`commit`* value saves time when the application needs more stack space, but increases the memory requirements and possibly the startup time. For ARM64, x86, and x64 machines, the default *`commit`* value is 4 KB.

Specify the *`reserve`* and *`commit`* values in decimal or C-language hexadecimal notation (use a `0x` prefix).

Another way to set the size of the stack is with the STACKSIZE statement in a module-definition (*`.def`*) file. `STACKSIZE` overrides the Stack Allocations (`/STACK`) option if both are specified. You can change the stack size after the *`.exe`* file is built by using the EDITBIN tool.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **System** property page.
3. Modify one of the following properties:
 - **Stack Commit Size**
 - **Stack Reserve Size**

To set this linker option programmatically

1. See StackCommitSize and StackReserveSize properties.

See also

MSVC linker reference
MSVC linker options

/STUB (MS-DOS Stub File Name)

Article • 08/03/2021

```
/STUB:filename
```

Arguments

filename
An MS-DOS application.

Remarks

The /STUB option attaches an MS-DOS stub program to a Win32 program.

A stub program is invoked if the file is executed in MS-DOS. It usually displays an appropriate message; however, any valid MS-DOS application can be a stub program.

Specify a *filename* for the stub program after a colon (:) on the command line. The linker checks *filename* and issues an error message if the file is not an executable. The program must be an .exe file; a .com file is invalid for a stub program.

If this option is not used, the linker attaches a default stub program that issues the following message:

```
This program cannot be run in MS-DOS mode.
```

When building a virtual device driver, *filename* allows the user to specify a file name that contains an IMAGE_DOS_HEADER structure (defined in WINNT.H) to be used in the VXD, rather than the default header.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.

3. Enter the option into the **Additional Options** box.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC Linker Options

`/SUBSYSTEM` (Specify Subsystem)

Article • 09/22/2022

Specify the Windows subsystem targeted by the executable.

Syntax

> `/SUBSYSTEM:` { `BOOT_APPLICATION` | `CONSOLE` | `EFI_APPLICATION` |
> `EFI_BOOT_SERVICE_DRIVER` | `EFI_ROM` | `EFI_RUNTIME_DRIVER` | `NATIVE` |
> `POSIX` | `WINDOWS` }
> [`,` *`major`* [`.` *`minor`*]]

Arguments

`BOOT_APPLICATION`

An application that runs in the Windows boot environment. For more information about boot applications, see About BCD.

`CONSOLE`

Win32 character-mode application. The operating system provides a console for console applications. If `main` or `wmain` is defined for native code, `int main(array<String ^> ^)` is defined for managed code, or you build the application completely by using `/clr:safe`, CONSOLE is the default.

`EFI_APPLICATION`

`EFI_BOOT_SERVICE_DRIVER`

`EFI_ROM`

`EFI_RUNTIME_DRIVER`

The Extensible Firmware Interface subsystems. For more information, see the UEFI specification. For examples, see the Intel UEFI Driver and Application Tool Resources. The minimum version and default version is 1.0.

`NATIVE`

Kernel mode drivers for Windows NT. This option is normally reserved for Windows system components. If /DRIVER:WDM is specified, `NATIVE` is the default.

`POSIX`

Application that runs with the POSIX subsystem in Windows NT.

`WINDOWS`
The application doesn't require a console, probably because it creates its own windows for interaction with the user. If `WinMain` or `wWinMain` is defined for native code, or `WinMain(HINSTANCE *, HINSTANCE *, char *, int)` or `wWinMain(HINSTANCE *, HINSTANCE *, wchar_t *, int)` is defined for managed code, `WINDOWS` is the default.

`major` and *`minor`*
(Optional) Specify the minimum required version of the subsystem. The arguments are decimal numbers in the range 0 through 65,535. There are no upper bounds for version numbers.

Remarks

The **`/SUBSYSTEM`** option specifies the environment for the executable.

The choice of subsystem affects the entry point symbol (or entry point function) that the linker will select.

The optional minimum and default *`major`* and *`minor`* version numbers for the subsystems are as follows:

Subsystem	Minimum	Default
`BOOT_APPLICATION`	1.0	1.0
`CONSOLE`	5.01 (x86) 5.02 (x64) 6.02 (ARM)	6.00 (x86, x64) 6.02 (ARM)
`WINDOWS`	5.01 (x86) 5.02 (x64) 6.02 (ARM)	6.00 (x86, x64) 6.02 (ARM)
`NATIVE` (with **`/DRIVER:WDM`**)	1.00 (x86) 1.10 (x64, ARM)	1.00 (x86) 1.10 (x64, ARM)
`NATIVE` (without **`/DRIVER:WDM`**)	4.00 (x86) 5.02 (x64) 6.02 (ARM)	4.00 (x86) 5.02 (x64) 6.02 (ARM)
`POSIX`	1.0	19.90
`EFI_APPLICATION`, `EFI_BOOT_SERVICE_DRIVER`, `EFI_ROM`, `EFI_RUNTIME_DRIVER`	1.0	1.0

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **System** property page.
3. Modify the **SubSystem** property.

To set this linker option programmatically

- See SubSystem.

See also

MSVC linker reference
MSVC linker options

/SWAPRUN (Load Linker Output to Swap File)

Article • 08/03/2021

```
/SWAPRUN:{NET|CD}
```

Remarks

The /SWAPRUN option tells the operating system to first copy the linker output to a swap file, and then run the image from there. This is a Windows NT 4.0 (and later) feature.

If NET is specified, the operating system will first copy the binary image from the network to a swap file and load it from there. This option is useful for running applications over the network. When CD is specified, the operating system will copy the image on a removable disk to a page file and then load it.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **System** property page.
3. Modify one of the following properties:
 - **Swap Run From CD**
 - **Swap Run From Network**

To set this linker option programmatically

1. See SwapRunFromCD and SwapRunFromNet properties.

See also

MSVC linker reference

MSVC Linker Options

/TLBID (Specify Resource ID for TypeLib)

Article • 08/03/2021

```
/TLBID:id
```

Arguments

id
A user-specified value for a linker-created type library. It overrides the default resource ID of 1.

Remarks

When compiling a program that uses attributes, the linker will create a type library. The linker will assign a resource ID of 1 to the type library.

If this resource ID conflicts with one of your existing resources, you can specify another ID with /TLBID. The range of values that you can pass to `id` is 1 to 65535.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Embedded IDL** property page.
3. Modify the **TypeLib Resource ID** property.

To set this linker option programmatically

1. See TypeLibraryResourceID.

See also

MSVC linker reference
MSVC Linker Options

/TLBOUT (Name .TLB File)

Article • 08/03/2021

```
/TLBOUT:[path\]filename
```

Arguments

path
An absolute or relative path specification for where the .tlb file should be created.

filename
Specifies the name of the .tlb file created by the MIDL compiler. No file extension is assumed; specify *filename*.tlb if you want a .tlb extension.

Remarks

The /TLBOUT option specifies the name and extension of the .tlb file.

The MIDL compiler is called by the MSVC linker when linking projects that have the module attribute.

If /TLBOUT is not specified, the .tlb file will get its name from /IDLOUT *filename*. If /IDLOUT is not specified, the .tlb file will be called vc70.tlb.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Embedded IDL** property page.
3. Modify the **Type Library** property.

To set this linker option programmatically

1. See TypeLibraryFile.

See also

MSVC linker reference
MSVC Linker Options
/IGNOREIDL (Don't Process Attributes into MIDL)
/MIDL (Specify MIDL Command Line Options)
Building an Attributed Program

`/TSAWARE` (Create Terminal Server aware application)

Article • 03/03/2022

> `/TSAWARE`[`:NO`]

Remarks

The `/TSAWARE` option sets a flag in the `IMAGE_OPTIONAL_HEADER` `DllCharacteristics` field in the program image's optional header. When this flag is set, Terminal Server won't make certain changes to the application.

When an application isn't Terminal Server aware (also known as a legacy application), Terminal Server makes certain modifications to the legacy application to make it work properly in a multiuser environment. For example, Terminal Server creates a virtual *`Windows`* folder, such that each user gets a *`Windows`* folder instead of getting the system's *`Windows`* directory. This virtual folder gives users access to their own INI files. In addition, Terminal Server makes some adjustments to the registry for a legacy application. These modifications slow the loading of the legacy application on Terminal Server.

If an application is Terminal Server aware, it must not rely on INI files or write to the `HKEY_CURRENT_USER` registry during setup.

If you use `/TSAWARE` and your application still uses INI files, the files will be shared by all users of the system. If that's acceptable, you can still link your application with `/TSAWARE`; otherwise you need to use `/TSAWARE:NO`.

The `/TSAWARE` option is enabled by default for Windows and console applications. For more information, see /SUBSYSTEM and /VERSION.

`/TSAWARE` isn't valid for drivers or DLLs.

If an application was linked with `/TSAWARE`, DUMPBIN /HEADERS will display information to that effect.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **System** property page.
3. Modify the **Terminal Server** property.

To set this linker option programmatically

- See TerminalServerAware.

See also

MSVC linker reference
MSVC Linker Options
Storing User-Specific Information
Legacy Applications in a Terminal Services Environment

/USEPROFILE (Run PGO in thread safe mode)

Article • 08/03/2021

This linker option together with /LTCG (Link-time code generation tells the linker to build by using profile-guided optimization (PGO) training data.

Syntax

> **/USEPROFILE**[:{**AGGRESSIVE**|**PGD**=*filename*}]

Arguments

AGGRESSIVE
This optional argument specifies that aggressive speed optimizations should be used during optimized code generation.

PGD=*filename*
Specifies a base file name for the .pgd file. By default, the linker uses the base executable file name with a .pgd extension.

Remarks

The **/USEPROFILE** linker option is used together with **/LTCG** to generate or update an optimized build based on PGO training data. It is the equivalent of the deprecated **/LTCG:PGUPDATE** and **/LTCG:PGOPTIMIZE** options.

The optional **AGGRESSIVE** argument disables size-related heuristics to attempt to optimize for speed. This may result in optimizations that substantially increase the size of your executable, and may not increase the resulting speed. You should profile and compare the results of using and not using **AGGRESSIVE**. This argument must be specified explicitly; it is not enabled by default.

The **PGD** argument specifies an optional name for the training data .pgd file to use, the same as in /GENPROFILE or /FASTGENPROFILE. It is the equivalent of the deprecated **/PGD** switch. By default, or if no *filename* is specified, a .pgd file that has the same base name as the executable is used.

The **/USEPROFILE** linker option is new in Visual Studio 2015.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Optimization** property page.
3. In the **Link Time Code Generation** property, choose **Use Link Time Code Generation (/LTCG)**.
4. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
5. Enter the **/USEPROFILE** option and optional arguments into the **Additional Options** box. Choose **OK** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

/GENPROFILE and /FASTGENPROFILE
/LTCG
Profile-Guided Optimizations
Environment Variables for Profile-Guided Optimizations

`/VERBOSE` (Print progress messages)

Article • 08/03/2021

Outputs progress messages during the link process.

Syntax

> `/VERBOSE`[`:`{`CLR`|`ICF`|`INCR`|`LIB`|`REF`|`SAFESEH`|`UNUSEDDELAYLOAD`|`UNUSEDLIBS`}]

Remarks

The linker sends information about the progress of the linking session to the **Output** window. On the command line, the information is sent to standard output, and can be redirected to a file.

Option	Description
`/VERBOSE`	Displays details about the linking process.
`/VERBOSE:CLR`	Displays information about linker activity specific to objects and metadata compiled by using /clr.
`/VERBOSE:ICF`	Displays information about linker activity that results from the use of /OPT:ICF.
`/VERBOSE:INCR`	Displays information about the incremental link process.
`/VERBOSE:LIB`	Displays progress messages that indicate just the libraries searched. The displayed information includes the library search process. It lists each library and object name (with full path), the symbol being resolved from the library, and a list of objects that reference the symbol.
`/VERBOSE:REF`	Displays information about linker activity that results from the use of /OPT:REF.
`/VERBOSE:SAFESEH`	Displays information about modules that are incompatible with safe structured exception handling when /SAFESEH isn't specified.
`/VERBOSE:UNUSEDDELAYLOAD`	Displays information about any delay loaded DLLs that have no symbols used when the image is created.
`/VERBOSE:UNUSEDLIBS`	Displays information about any library files that are unused when the image is created.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Add the option to the **Additional Options** box.

To set this linker option programmatically

- See ShowProgress.

See also

MSVC linker reference
MSVC Linker Options

`/VERSION` (Version information)

Article • 03/03/2022

> `/VERSION:`*`major`*[`.`*`minor`*]

Arguments

`major` and *`minor`*
The version number you want in the header of the EXE or DLL file.

Remarks

The `/VERSION` option tells the linker to put a version number in the header of the EXE or DLL file. Use DUMPBIN /HEADERS to see the image version field of the `OPTIONAL HEADER VALUES` to see the effect of `/VERSION`.

The *`major`* and *`minor`* arguments are decimal numbers in the range 0 through 65,535. The default is version `0.0`.

The information specified with `/VERSION` doesn't affect the version information that appears for an application when you view its properties in File Explorer. That version information comes from a resource file that's used to build the application. For more information, see Version Information Editor.

Another way to insert a version number is with the VERSION module-definition statement.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Version** property.

To set this linker option programmatically

- See Version.

See also

MSVC linker reference
MSVC linker options

/WHOLEARCHIVE (Include All Library Object Files)

Article • 08/03/2021

Force the linker to include all object files in the static library in the linked executable.

Syntax

> **/WHOLEARCHIVE**
> **/WHOLEARCHIVE:***library*

Arguments

library
An optional pathname to a static library. The linker includes every object file from this library.

Remarks

The /WHOLEARCHIVE option forces the linker to include every object file from either a specified static library, or if no library is specified, from all static libraries specified to the LINK command. To specify the /WHOLEARCHIVE option for multiple libraries, you can use more than one /WHOLEARCHIVE switch on the linker command line. By default, the linker includes object files in the linked output only if they export symbols referenced by other object files in the executable. The /WHOLEARCHIVE option makes the linker treat all object files archived in a static library as if they were specified individually on the linker command line.

The /WHOLEARCHIVE option can be used to re-export all the symbols from a static library. This allows you to make sure that all of your library code, resources, and metadata are included when you create a component from more than one static library. If you see warning LNK4264 when you create a static library that contains Windows Runtime components for export, use the /WHOLEARCHIVE option when linking that library into another component or app.

The /WHOLEARCHIVE option was introduced in Visual Studio 2015 Update 2.

To set this linker option in Visual Studio

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. Add the `/WHOLEARCHIVE` option to the **Additional Options** text box.

See also

MSVC linker reference
MSVC Linker Options

/WINMD (Generate Windows Metadata)

Article • 08/03/2021

Enables generation of a Windows Runtime Metadata (.winmd) file.

> **/WINMD**[:{**NO**|**ONLY**}]

Arguments

/WINMD
The default setting for Universal Windows Platform apps. The linker generates both the binary executable file and the .winmd metadata file.

/WINMD:NO
The linker generates only the binary executable file, but not a .winmd file.

/WINMD:ONLY
The linker generates only the .winmd file, but not the binary executable file.

Remarks

The **/WINMD** linker option is used for UWP apps and Windows runtime components to control the creation of a Windows Runtime metadata (.winmd) file. A .winmd file is a kind of DLL that contains metadata for Windows runtime types and, in the case of runtime components, the implementations of those types. The metadata follows the ECMA-335 standard.

By default, the output file name has the form *binaryname*.winmd. To specify a different file name, use the /WINMDFILE option.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Windows Metadata** property page.
3. In the **Generate Windows Metadata** drop-down list box, select the option you want.

See also

Walkthrough: Creating a Simple Windows Runtime component and calling it from JavaScript
Introduction to Microsoft Interface Definition Language 3.0
/WINMDFILE (Specify winmd File)
/WINMDKEYFILE (Specify winmd Key File)
/WINMDKEYCONTAINER (Specify Key Container)
/WINMDDELAYSIGN (Partially Sign a winmd)
MSVC linker reference
MSVC Linker Options

/WINMDFILE (Specify winmd File)

Article • 08/03/2021

Specifies the file name for the Windows Runtime Metadata (.winmd) output file that is generated by the /WINMD linker option.

```
/WINMDFILE:filename
```

Remarks

Use the value that is specified in `filename` to override the default .winmd file name (`binaryname`.winmd). Notice that you do not append ".winmd" to `filename`. If multiple values are listed on the **/WINMDFILE** command line, the last one takes precedence.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Windows Metadata** property page.
3. In the **Windows Metadata File** box, enter the file location.

See also

/WINMD (Generate Windows Metadata)
MSVC linker reference
MSVC Linker Options

/WINMDKEYFILE (Specify winmd Key File)

Article • 08/03/2021

Specifies a key or a key pair to sign a Windows Runtime Metadata (.winmd) file.

```
/WINMDKEYFILE:filename
```

Remarks

Resembles the /KEYFILE linker option that is applied to a .winmd file.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Windows Metadata** property page.
3. In the **Windows Metadata Key File** box, enter the file location.

See also

MSVC linker reference
MSVC Linker Options

/WINMDKEYCONTAINER (Specify Key Container)

Article • 08/03/2021

Specifies a key container to sign a Windows Metadata (.winmd) file.

```
/WINMDKEYCONTAINER:name
```

Remarks

Resembles the /KEYCONTAINER linker option that is applied to a (.winmd) file.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Windows Metadata** property page.
3. In the **Windows Metadata Key Container** box, enter the location.

See also

MSVC linker reference
MSVC Linker Options

/WINMDDELAYSIGN (Partially Sign a winmd)

Article • 08/03/2021

Enables partial signing of a Windows Runtime Metadata (.winmd) file by putting the public key in the file.

```
/WINMDDELAYSIGN[:NO]
```

Remarks

Resembles the /DELAYSIGN linker option that is applied to the .winmd file. Use **/WINMDDELAYSIGN** if you want to put only the public key in the .winmd file. By default, the linker acts as if **/WINMDDELAYSIGN:NO** were specified; that is, it does not sign the winmd file.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Windows Metadata** property page.
3. In the **Windows Metadata Delay Sign** drop-down list box, select the option you want.

See also

MSVC linker reference
MSVC Linker Options

`/WX` (Treat linker warnings as errors)

Article • 02/06/2023

Specifies whether to treat linker warnings as errors.

Syntax

> `/WX`[`:NO`]
> `/WX`[`:`*nnnn*[`,`*nnnn*...]]

Remarks

The `/WX` linker option causes no output file to be generated if the linker generates a warning.

This option is similar to `/WX` for the compiler. For more information, see /w, /W0, /W1, /W2, /W3, /W4, /w1, /w2, /w3, /w4, /Wall, /wd, /we, /wo, /Wv, /WX (Warning Level). However, specifying `/WX` for the compilation doesn't imply that `/WX` will also be in effect for the link phase; you must explicitly specify `/WX` for each tool.

In Visual Studio 2022 and later versions, you can specify `/WX` with one or more comma-separated *nnnn* arguments, where *nnnn* is a number between 4000 and 4999. The linker treats the corresponding *LNKnnnn* warnings as errors.

By default, `/WX` isn't in effect. To treat linker warnings as errors, specify a `/WX` option. `/WX:NO` is the same as not specifying `/WX`, and overrides any previous `/WX` linker option.

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For more information, see Set compiler and build properties.
2. To set or unset all warnings as errors, select the **Configuration Properties** > **Linker** > **General** property page.
3. Modify the **Treat Linker Warnings as Errors** property.
4. To set specific warnings as errors, select the **Configuration Properties** > **Linker** > **Command Line** property page.

5. In the **Additional Options** edit control, add `/WX:warnings`, where `warnings` is a comma-separated list of linker warning numbers.

6. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- See AdditionalOptions.

See also

MSVC linker reference
MSVC linker options
/WX compiler option

Decorated names

Article • 06/15/2022

Functions, data, and objects in C and C++ programs are represented internally by their decorated names. A *decorated name* is an encoded string created by the compiler during compilation of an object, data, or function definition. It records calling conventions, types, function parameters and other information together with the name. This name decoration, also known as *name mangling*, helps the linker find the correct functions and objects when linking an executable.

The decorated naming conventions have changed in various versions of Visual Studio, and can also be different on different target architectures. To link correctly with source files created by using Visual Studio, C and C++ DLLs and libraries should be compiled by using the same compiler toolset, flags, and target architecture.

> **Note**
>
> Libraries built by Visual Studio 2015 or later can be consumed by applications built with later versions of Visual Studio through Visual Studio 2022. For more information, see **C++ binary compatibility between Visual Studio versions**.

Using decorated names

Normally, you don't have to know the decorated name to write code that compiles and links successfully. Decorated names are an implementation detail internal to the compiler and linker. The tools can usually handle the name in its undecorated form. However, a decorated name is sometimes required when you specify a function name to the linker and other tools. For example, to match overloaded C++ functions, members of namespaces, class constructors, destructors and special member functions, you must specify the decorated name. For details about the option flags and other situations that require decorated names, see the documentation for the tools and options that you're using.

If you change the function name, class, calling convention, return type, or any parameter, the decorated name also changes. In this case, you must get the new decorated name and use it everywhere the decorated name is specified.

Name decoration is also important when linking to code written in other programming languages or using other compilers. Different compilers use different name decoration conventions. When your executable links to code written in another language, special

care must be taken to match the exported and imported names and calling conventions. Assembly language code must use the MSVC decorated names and calling conventions to link to source code written using MSVC.

Format of a C++ decorated name

A decorated name for a C++ function contains the following information:

- The function name.
- The class that the function is a member of, if it's a member function. The decoration may include the class that encloses the class that contains the function, and so on.
- The namespace the function belongs to, if it's part of a namespace.
- The types of the function parameters.
- The calling convention.
- The return type of the function.
- An optional target-specific element. In ARM64EC objects, a `$$h` tag is inserted into the name.

The function and class names are encoded in the decorated name. The rest of the decorated name is a code that has internal meaning only for the compiler and the linker. The following are examples of undecorated and decorated C++ names.

Undecorated name	Decorated name
`int a(char){int i=3;return i;};`	`?a@@YAHD@Z`
`void __stdcall b::c(float){};`	`?c@b@@AAGXM@Z`

Format of a C decorated name

The form of decoration for a C function depends on the calling convention used in its declaration, as shown in the following table. It's also the decoration format that's used when C++ code is declared to have `extern "C"` linkage. The default calling convention is `__cdecl`. In a 64-bit environment, C or `extern "C"` functions are only decorated when using the `__vectorcall` calling convention.

Calling convention	Decoration
`__cdecl`	Leading underscore (`_`)
`__stdcall`	Leading underscore (`_`) and a trailing at sign (`@`) followed by the number of bytes in the parameter list in decimal
`__fastcall`	Leading and trailing at signs (`@`) followed by a decimal number representing the number of bytes in the parameter list
`__vectorcall`	Two trailing at signs (`@@`) followed by a decimal number of bytes in the parameter list

For ARM64EC functions with C linkage (whether compiled as C or by using `extern "C"`), a `#` is prepended to the decorated name.

View decorated names

You can get the decorated form of a symbol name after you compile the source file that contains the data, object, or function definition or prototype. To examine decorated names in your program, you can use one of the following methods:

To use a listing to view decorated names

1. Generate a listing by compiling the source file that contains the data, object, or function definition or prototype with the /FA (Listing file type) compiler option set to assembly with source code (`/FAs`).

 For example, enter `cl /c /FAs example.cpp` at a developer command prompt to generate a listing file, *`example.asm`*.

2. In the resulting listing file, find the line that starts with `PUBLIC` and ends a semicolon (`;`) followed by the undecorated data or function name. The symbol between `PUBLIC` and the semicolon is the decorated name.

To use DUMPBIN to view decorated names

1. To see the exported symbols in an OBJ or LIB file, enter `dumpbin /exports <obj-or-lib-file>` at a developer command prompt.

2. To find the decorated form of a symbol, look for the undecorated name in parentheses. The decorated name is on the same line, before the undecorated

name.

Viewing undecorated names

You can use undname.exe to convert a decorated name to its undecorated form. This example shows how it works:

Windows Command Prompt

```
C:\>undname ?func1@a@@AAEXH@Z
Microsoft (R) C++ Name Undecorator
Copyright (C) Microsoft Corporation. All rights reserved.

Undecoration of :- "?func1@a@@AAEXH@Z"
is :- "private: void __thiscall a::func1(int)"
```

See also

Additional MSVC build tools
Using extern to specify linkage

Module-Definition (.Def) Files

Article • 08/03/2021

Module-definition (.def) files provide the linker with information about exports, attributes, and other information about the program to be linked. A .def file is most useful when building a DLL. Because there are MSVC Linker Options that can be used instead of module-definition statements, .def files are generally not necessary. You can also use __declspec(dllexport) as a way to specify exported functions.

You can invoke a .def file during the linker phase with the /DEF (Specify Module-Definition File) linker option.

If you are building an .exe file that has no exports, using a .def file will make your output file larger and slower loading.

For an example, see Exporting from a DLL Using DEF Files.

See the following sections for more information:

- Rules for Module-Definition Statements
- EXPORTS
- HEAPSIZE
- LIBRARY
- NAME
- SECTIONS
- STACKSIZE
- STUB
- VERSION
- Reserved words

See also

C/C++ Building Reference
MSVC Linker Options

Rules for Module-Definition Statements

Article • 08/03/2021

The following syntax rules apply to all statements in a .def file. Other rules that apply to specific statements are described with each statement.

- Statements, attribute keywords, and user-specified identifiers are case sensitive.
- Long file names containing spaces or semicolons (;) must be enclosed in quotation marks (").
- Use one or more spaces, tabs, or newline characters to separate a statement keyword from its arguments and to separate statements from each other. A colon (:) or equal sign (=) that designates an argument is surrounded by zero or more spaces, tabs, or newline characters.
- A **NAME** or **LIBRARY** statement, if used, must precede all other statements.
- The **SECTIONS** and **EXPORTS** statements can appear more than once in the .def file. Each statement can take multiple specifications, which must be separated by one or more spaces, tabs, or newline characters. The statement keyword must appear once before the first specification and can be repeated before each additional specification.
- Many statements have an equivalent LINK command-line option. See the description of the corresponding LINK option for additional details.
- Comments in the .def file are designated by a semicolon (;) at the beginning of each comment line. A comment cannot share a line with a statement, but it can appear between specifications in a multiline statement. (**SECTIONS** and **EXPORTS** are multiline statements.)
- Numeric arguments are specified in base 10 or hexadecimal.
- If a string argument matches a reserved word, it must be enclosed in double quotation marks (").

See also

Module-Definition (.Def) Files

EXPORTS

Article • 08/03/2021

Introduces a section of one or more export definitions that specify the exported names or ordinals of functions or data. Each definition must be on a separate line.

```
DEF
```

```
EXPORTS
   definition
```

Remarks

The first *definition* can be on the same line as the `EXPORTS` keyword or on a subsequent line. The .DEF file can contain one or more `EXPORTS` statements.

The syntax for an export *definition* is:

> *entryname*[=*internal_name*|*other_module.exported_name*] [@*ordinal* [**NONAME**]] [[**PRIVATE**] | [**DATA**]]

entryname is the function or variable name that you want to export. This is required. If the name that you export differs from the name in the DLL, specify the export's name in the DLL by using *internal_name*. For example, if your DLL exports a function `func1` and you want callers to use it as `func2`, you would specify:

```
DEF
```

```
EXPORTS
   func2=func1
```

If the name that you export is from some other module, specify the export's name in the DLL by using *other_module.exported_name*. For example, if your DLL exports a function `other_module.func1` and you want callers to use it as `func2`, you would specify:

```
DEF
```

```
EXPORTS
   func2=other_module.func1
```

If the name that you export is from another module that exports by ordinal, specify the export's ordinal in the DLL by using *other_module*.#*ordinal*. For example, if your DLL exports a function from the other module where it is ordinal 42, and you want callers to use it as `func2`, you would specify:

```
DEF

EXPORTS
   func2=other_module.#42
```

Because the MSVC compiler uses name decoration for C++ functions, you must either use the decorated name *internal_name* or define the exported functions by using `extern "C"` in the source code. The compiler also decorates C functions that use the __stdcall calling convention with an underscore (_) prefix and a suffix composed of the at sign (@) followed by the number of bytes (in decimal) in the argument list.

To find the decorated names produced by the compiler, use the DUMPBIN tool or the linker /MAP option. The decorated names are compiler-specific. If you export the decorated names in the .DEF file, executables that link to the DLL must also be built by using the same version of the compiler. This ensures that the decorated names in the caller match the exported names in the .DEF file.

You can use @*ordinal* to specify that a number, and not the function name, goes into the DLL's export table. Many Windows DLLs export ordinals to support legacy code. It was common to use ordinals in 16-bit Windows code, because it can help minimize the size of a DLL. We don't recommend exporting functions by ordinal unless your DLL's clients need it for legacy support. Because the .LIB file will contain the mapping between the ordinal and the function, you can use the function name as you normally would in projects that use the DLL.

By using the optional **NONAME** keyword, you can export by ordinal only and reduce the size of the export table in the resulting DLL. However, if you want to use GetProcAddress on the DLL, you must know the ordinal because the name will not be valid.

The optional keyword **PRIVATE** prevents *entryname* from being included in the import library generated by LINK. It does not affect the export in the image also generated by LINK.

The optional keyword **DATA** specifies that an export is data, not code. This example shows how you could export a data variable named `exported_global`:

```
DEF
```

```
EXPORTS
   exported_global DATA
```

There are four ways to export a definition, listed in recommended order:

1. The __declspec(dllexport) keyword in the source code
2. An `EXPORTS` statement in a .DEF file
3. An /EXPORT specification in a LINK command
4. A comment directive in the source code, of the form `#pragma comment(linker, "/export: definition ")`. The following example shows a #pragma comment directive before a function declaration, where `PlainFuncName` is the undecorated name, and `_PlainFuncName@4` is the decorated name of the function:

 C++

   ```
   #pragma comment(linker, "/export:PlainFuncName=_PlainFuncName@4")
   BOOL CALLBACK PlainFuncName( Things * lpParams)
   ```

The #pragma directive is useful if you need to export an undecorated function name, and have different exports depending on the build configuration (for example, in 32-bit or 64-bit builds).

All four methods can be used in the same program. When LINK builds a program that contains exports, it also creates an import library, unless an .EXP file is used in the build.

Here's an example of an EXPORTS section:

DEF

```
EXPORTS
   DllCanUnloadNow      @1          PRIVATE
   DllWindowName = WindowName       DATA
   DllGetClassObject    @4 NONAME   PRIVATE
   DllRegisterServer    @7
   DllUnregisterServer
```

When you export a variable from a DLL by using a .DEF file, you do not have to specify `__declspec(dllexport)` on the variable. However, in any file that uses the DLL, you must still use `__declspec(dllimport)` on the declaration of data.

See also

Rules for Module-Definition Statements

LIBRARY

Article • 08/03/2021

Tells LINK to create a DLL. At the same time, LINK creates an import library, unless an .exp file is used in the build.

```
LIBRARY [library][BASE=address]
```

Remarks

The *library* argument specifies the name of the DLL. You can also use the /OUT linker option to specify the DLL's output name.

The BASE=*address* argument sets the base address that the operating system uses to load the DLL. This argument overrides the default DLL location of 0x10000000. See the description of the /BASE option for details about base addresses.

Remember to use the /DLL linker option when you build a DLL.

See also

Rules for Module-Definition Statements

HEAPSIZE

Article • 08/03/2021

Exposes the same functionality as the /HEAP linker option.

```
/HEAP:reserve[,commit]
```

See also

Rules for Module-Definition Statements

NAME (C/C++)

Article • 08/03/2021

Specifies a name for the main output file.

```
NAME [application][BASE=address]
```

Remarks

An equivalent way to specify an output file name is with the /OUT linker option, and an equivalent way to set the base address is with the /BASE linker option. If both are specified, /OUT overrides **NAME**.

If you build a DLL, NAME will only affect the DLL name.

See also

Rules for Module-Definition Statements

SECTIONS (C/C++)

Article • 08/03/2021

Introduces a section of one or more `definitions` that are access specifiers on sections in your project's output file.

```
SECTIONS
definitions
```

Remarks

Each definition must be on a separate line. The `SECTIONS` keyword can be on the same line as the first definition or on a preceding line. The .def file can contain one or more `SECTIONS` statements.

This `SECTIONS` statement sets attributes for one or more sections in the image file, and can be used to override the default attributes for each type of section.

The format for `definitions` is:

`.section_name specifier`

where `.section_name` is the name of a section in your program image and `specifier` is one or more of the following access modifiers:

Modifier	Description
`EXECUTE`	The section is executable
`READ`	Allows read operations on data
`SHARED`	Shares the section among all processes that load the image
`WRITE`	Allows write operations on data

Separate specifier names with a space. For example:

```
SECTIONS
.rdata READ WRITE
```

`SECTIONS` marks the beginning of a list of section `definitions`. Each `definition` must be on a separate line. The `SECTIONS` keyword can be on the same line as the first `definition` or on a preceding line. The .def file can contain one or more `SECTIONS` statements. The `SEGMENTS` keyword is supported as a synonym for `SECTIONS`.

Older versions of Visual C++ supported:

```
section [CLASS 'classname'] specifier
```

The `CLASS` keyword is supported for compatibility, but is ignored.

An equivalent way to specify section attributes is with the /SECTION option.

See also

Rules for Module-Definition Statements

STACKSIZE

Article • 08/03/2021

Sets the size of the stack in bytes.

```
STACKSIZE reserve[,commit]
```

Remarks

An equivalent way to set the stack is with the Stack Allocations (/STACK) option. See the documentation on that option for details about the *reserve* and `commit` arguments.

This option has no effect on DLLs.

See also

Rules for Module-Definition Statements

STUB

Article • 08/03/2021

When used in a module definition file that builds a virtual device driver (VxD), allows you to specify a file name that contains an IMAGE_DOS_HEADER structure (defined in WINNT.H) to be used in the virtual device driver (VxD), rather than the default header.

```
STUB:filename
```

Remarks

An equivalent way to specify *filename* is with the /STUB linker option.

STUB is valid in a module definition file only when building a VxD.

See also

Rules for Module-Definition Statements

VERSION (C/C++)

Article • 08/03/2021

Tells LINK to put a number in the header of the .exe file or DLL.

```
VERSION major[.minor]
```

Remarks

The *major* and *minor* arguments are decimal numbers in the range 0 through 65,535. The default is version 0.0.

An equivalent way to specify a version number is with the Version Information (/VERSION) option.

See also

Rules for Module-Definition Statements

Linker support for delay-loaded DLLs

Article • 08/03/2021

The MSVC linker supports the delayed loading of DLLs. This feature relieves you of the need to use the Windows SDK functions LoadLibrary and GetProcAddress to implement DLL delayed loading.

Without delayed load, the only way to load a DLL at run time is by using `LoadLibrary` and `GetProcAddress`; the operating system loads the DLL when the executable or DLL using it gets loaded.

With delayed load, when you implicitly link a DLL, the linker provides options to delay the DLL load until the program calls a function in that DLL.

An application can delay load a DLL using the /DELAYLOAD (Delay load import) linker option with a helper function. (A default helper function implementation is provided by Microsoft.) The helper function loads the DLL on demand at runtime by calling `LoadLibrary` and `GetProcAddress` for you.

Consider delay loading a DLL if:

- Your program might not call a function in the DLL.
- A function in the DLL might not get called until late in your program's execution.

The delayed loading of a DLL can be specified during the build of either an EXE or DLL project. A DLL project that delays the loading of one or more DLLs itself shouldn't call a delay-loaded entry point in `DllMain`.

Specify DLLs to delay load

You can specify which DLLs to delay load by using the /delayload:*`dllname`* linker option. If you don't plan to use your own version of a helper function, you must also link your program with *`delayimp.lib`* (for desktop applications) or *`dloadhelper.lib`* (for UWP apps).

Here's a simple example of delay loading a DLL:

C++

```
// cl t.cpp user32.lib delayimp.lib  /link /DELAYLOAD:user32.dll
#include <windows.h>
// uncomment these lines to remove .libs from command line
```

```
// #pragma comment(lib, "delayimp")
// #pragma comment(lib, "user32")

int main() {
   // user32.dll will load at this point
   MessageBox(NULL, "Hello", "Hello", MB_OK);
}
```

Build the DEBUG version of the project. Step through the code using the debugger and you'll notice that *`user32.dll`* is loaded only when you make the call to `MessageBox`.

Explicitly unload a delay-loaded DLL

The /delay:unload linker option allows your code to explicitly unload a DLL that was delay loaded. By default, the delay-loaded imports remain in the import address table (IAT). However, if you use **`/delay:unload`** on the linker command line, the helper function supports the explicit unloading of the DLL by a `__FUnloadDelayLoadedDLL2` call, and resets the IAT to its original form. The now-invalid pointers get overwritten. The IAT is a field in the ImgDelayDescr structure that contains the address of a copy of the original IAT, if one exists.

Example of unloading a delay-loaded DLL

This example shows how to explicitly unload a DLL, *`MyDll.dll`*, which contains a function `fnMyDll`:

C

```
// link with /link /DELAYLOAD:MyDLL.dll /DELAY:UNLOAD
#include <windows.h>
#include <delayimp.h>
#include "MyDll.h"
#include <stdio.h>

#pragma comment(lib, "delayimp")
#pragma comment(lib, "MyDll")
int main()
{
    BOOL TestReturn;
    // MyDLL.DLL will load at this point
    fnMyDll();

    //MyDLL.dll will unload at this point
    TestReturn = __FUnloadDelayLoadedDLL2("MyDll.dll");

    if (TestReturn)
        printf_s("\nDLL was unloaded");
```

```
    else
        printf_s("\nDLL was not unloaded");
}
```

Important notes on unloading a delay-loaded DLL:

- You can find the implementation of the `__FUnloadDelayLoadedDLL2` function in the file *`delayhlp.cpp`*, in the MSVC *`include`* directory. For more information, see Understand the delay load helper function.
- The *`name`* parameter of the `__FUnloadDelayLoadedDLL2` function must exactly match (including case) what the import library contains. (That string is also in the import table in the image.) You can view the contents of the import library by using DUMPBIN /DEPENDENTS. If you prefer a case-insensitive string match, you can update `__FUnloadDelayLoadedDLL2` to use one of the case-insensitive CRT string functions, or a Windows API call.

Bind delay-loaded imports

The default linker behavior is to create a bindable import address table (IAT) for the delay-loaded DLL. If the DLL is bound, the helper function attempts to use the bound information instead of calling `GetProcAddress` on each of the referenced imports. If either the timestamp or the preferred address doesn't match the one in the loaded DLL, the helper function assumes the bound import address table is out of date. It proceeds as if the IAT doesn't exist.

If you never intend to bind the delay-loaded imports of a DLL, specify /delay:nobind on the linker command line. The linker won't generate the bound import address table, which saves space in the image file.

Load all imports for a delay-loaded DLL

The `__HrLoadAllImportsForDll` function, which is defined in *`delayhlp.cpp`*, tells the linker to load all imports from a DLL that was specified with the /delayload linker option.

When you load all imports at once, you can centralize error handling in one place. You can avoid structured exception handling around all the actual calls to the imports. It also avoids a situation where your application fails part way through a process: For example, if the helper code fails to load an import, after successfully loading others.

Calling `__HrLoadAllImportsForDll` doesn't change the behavior of hooks and error handling. For more information, see Error handling and notification.

`__HrLoadAllImportsForDll` makes a case-sensitive comparison to the name stored inside the DLL itself.

Here's an example that uses `__HrLoadAllImportsForDll` in a function called `TryDelayLoadAllImports` to attempt to load a named DLL. It uses a function, `CheckDelayException`, to determine exception behavior.

C

```
int CheckDelayException(int exception_value)
{
    if (exception_value == VcppException(ERROR_SEVERITY_ERROR,
ERROR_MOD_NOT_FOUND) ||
        exception_value == VcppException(ERROR_SEVERITY_ERROR,
ERROR_PROC_NOT_FOUND))
    {
        // This example just executes the handler.
        return EXCEPTION_EXECUTE_HANDLER;
    }
    // Don't attempt to handle other errors
    return EXCEPTION_CONTINUE_SEARCH;
}

bool TryDelayLoadAllImports(LPCSTR szDll)
{
    __try
    {
        HRESULT hr = __HrLoadAllImportsForDll(szDll);
        if (FAILED(hr))
        {
            // printf_s("Failed to delay load functions from %s\n", szDll);
            return false;
        }
    }
    __except (CheckDelayException(GetExceptionCode()))
    {
        // printf_s("Delay load exception for %s\n", szDll);
        return false;
    }
    // printf_s("Delay load completed for %s\n", szDll);
    return true;
}
```

You can use the result of `TryDelayLoadAllImports` to control whether you call the import functions or not.

Error handling and notification

If your program uses delay-loaded DLLs, it must handle errors robustly. Failures that occur while the program is running will result in unhandled exceptions. For more information on DLL delay load error handling and notification, see Error handling and notification.

Dump delay-loaded imports

Delay-loaded imports can be dumped by using DUMPBIN /IMPORTS. These imports show up with slightly different information than standard imports. They're segregated into their own section of the `/imports` list, and are explicitly labeled as delay-loaded imports. If there's unload information present in the image, that is noted. If there's bind information present, the time and date stamp of the target DLL is noted along with the bound addresses of the imports.

Constraints on delay-load DLLs

There are several constraints on the delay loading of DLL imports.

- Imports of data can't be supported. A workaround is to explicitly handle the data import yourself by using `LoadLibrary` (or by using GetModuleHandle after you know the delay-load helper has loaded the DLL) and `GetProcAddress`.
- Delay loading *`Kernel32.dll`* isn't supported. This DLL must be loaded for the delay-load helper routines to work.
- Binding of forwarded entry points isn't supported.
- A process may have different behavior if a DLL is delay-loaded, instead of loaded on start-up. It can be seen if there are per-process initializations that occur in the entry point of the delay-loaded DLL. Other cases include static TLS (thread local storage), declared using __declspec(thread), which isn't handled when the DLL is loaded via `LoadLibrary`. Dynamic TLS, using TlsAlloc, TlsFree, TlsGetValue, and TlsSetValue, is still available for use in either static or delay-loaded DLLs.
- Reinitialize static global function pointers to imported functions after the first call of each function. That's required because the first use of a function pointer points to the thunk, not the loaded function.
- There's currently no way to delay the loading of only specific procedures from a DLL while using the normal import mechanism.

- Custom calling conventions (such as using condition codes on x86 architectures) aren't supported. Also, the floating-point registers aren't saved on any platform. Take care if your custom helper routine or hook routines use floating-point types: The routines must save and restore the complete floating-point state on machines that use register calling conventions with floating-point parameters. Be careful about delay-loading the CRT DLL, particularly if you call CRT functions that take floating-point parameters on a numeric data processor (NDP) stack in the help function.

Understand the delay load helper function

The helper function for linker-supported delayed loading is what actually loads the DLL at runtime. You can modify the helper function to customize its behavior. Instead of using the supplied helper function in `delayimp.lib`, write your own function and link it to your program. One helper function serves all delay loaded DLLs. For more information, see Understand the delay load helper function and Develop your own helper function.

See also

Create C/C++ DLLs in Visual Studio
MSVC linker reference

Error handling and notification

Article • 08/03/2021

If your program uses delay-loaded DLLs, it must handle errors robustly, since failures that occur while the program is running will result in unhandled exceptions. Failure handling is composed of two portions: Recovery through a hook, and reporting via an exception.

For more information on DLL delay load error handling and notification, see Understand the helper function.

For more information on hook functions, see Structure and constant definitions.

Recovery through a hook

Your code may need to recover on a failure, or to provide an alternate library or routine. You can provide a hook to the helper function that can supply the alternative code, or remedy the situation. The hook routine needs to return a suitable value, so that processing can continue (an `HINSTANCE` or `FARPROC`). Or, it can return 0 to indicate that an exception should be thrown. It could also throw its own exception or `longjmp` out of the hook. There are notification hooks and failure hooks. The same routine may be used for both.

Notification hooks

The delay load notification hooks are called just before the following actions are taken in the helper routine:

- The stored handle to the library is checked to see if it has already been loaded.
- `LoadLibrary` is called to attempt the load of the DLL.
- `GetProcAddress` is called to attempt to get the address of the procedure.
- Return to the delay import load thunk.

The notification hook is enabled:

- By supplying a new definition of the pointer `__pfnDliNotifyHook2` that's initialized to point to your own function that receives the notifications.

 -or-

- By setting the pointer `__pfnDliNotifyHook2` to your hook function before any calls to the DLL that the program is delay loading.

If the notification is `dliStartProcessing`, the hook function can return:

- `NULL`

 The default helper handles the loading of the DLL. It's useful to call just for informational purposes.

- a function pointer

 Bypass the default delay-load handling. It lets you supply your own load handler.

If the notification is `dliNotePreLoadLibrary`, the hook function can return:

- 0, if it just wants informational notifications.
- The `HMODULE` for the loaded DLL, if it loaded the DLL itself.

If the notification is `dliNotePreGetProcAddress`, the hook function can return:

- 0, if it just wants informational notifications.
- The imported function's address, if the hook function gets the address itself.

If the notification is `dliNoteEndProcessing`, the hook function's return value is ignored.

If this pointer is initialized (nonzero), the delay load helper invokes the function at certain notification points throughout its execution. The function pointer has the following definition:

C

```
// The "notify hook" gets called for every call to the
// delay load helper.  This allows a user to hook every call and
// skip the delay load helper entirely.
//
// dliNotify == {
//  dliStartProcessing |
//  dliNotePreLoadLibrary  |
//  dliNotePreGetProc |
//  dliNoteEndProcessing}
//  on this call.
//
ExternC
PfnDliHook   __pfnDliNotifyHook2;

// This is the failure hook, dliNotify = {dliFailLoadLib|dliFailGetProc}
```

```
ExternC
PfnDliHook   __pfnDliFailureHook2;
```

The notifications pass in a `DelayLoadInfo` structure to the hook function along with the notification value. This data is identical to the data used by the delay load helper routine. The notification value will be one of the values defined in Structure and constant definitions.

Failure hooks

The failure hook is enabled in the same manner as the notification hook. The hook routine needs to return a suitable value so that processing can continue (an `HINSTANCE` or `FARPROC`), or 0 to indicate that an exception should be thrown.

The pointer variable that refers to the user-defined function is:

```
C

// This is the failure hook, dliNotify = {dliFailLoadLib|dliFailGetProc}
ExternC
PfnDliHook   __pfnDliFailureHook2;
```

The **`DelayLoadInfo`** structure contains all the pertinent data necessary for detailed reporting of the error, including the value from `GetLastError`.

If the notification is **`dliFailLoadLib`**, the hook function can return:

- 0, if it can't handle the failure.
- An `HMODULE`, if the failure hook fixed the problem and loaded the library itself.

If the notification is **`dliFailGetProc`**, the hook function can return:

- 0, if it can't handle the failure.
- A valid proc address (import function address), if the failure hook succeeded in getting the address itself.

Report by using an exception

If all that's required to handle the error is to abort the procedure, no hook is necessary, as long as the user code can handle the exception.

Delay load exception codes

Structured exception codes can be raised when failures occur during a delayed load. The exception values are specified by using a `VcppException` macro:

```
C
```

```
//
// Exception information
//
#define FACILITY_VISUALCPP  ((LONG)0x6d)
#define VcppException(sev,err)  ((sev) | (FACILITY_VISUALCPP<<16) | err)
```

For a `LoadLibrary` failure, the standard `VcppException(ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND)` is thrown. For a `GetProcAddress` failure, the error thrown is `VcppException(ERROR_SEVERITY_ERROR, ERROR_PROC_NOT_FOUND)`. The exception passes a pointer to a `DelayLoadInfo` structure. It's in the `LPDWORD` value retrieved by `GetExceptionInformation` from the EXCEPTION_RECORD structure, in the `ExceptionInformation[0]` field.

If the incorrect bits are set in the `grAttrs` field, the exception `ERROR_INVALID_PARAMETER` is thrown. This exception is, for all intents and purposes, fatal.

For more information, see Structure and constant definitions.

See also

Linker support for delay-loaded DLLs

Understand the delay load helper function

Article • 08/03/2021

The helper function for linker-supported delayed loading is what actually loads the DLL at runtime. You can modify the helper function to customize its behavior. Instead of using the supplied helper function in *`delayimp.lib`*, write your own function and link it to your program. One helper function serves all delay loaded DLLs.

You can provide your own version of the helper function if you want to do specific processing based on the names of the DLL or imports.

The helper function takes these actions:

- Checks the stored handle to the library to see if it has already been loaded
- Calls `LoadLibrary` to attempt to load the DLL
- Calls `GetProcAddress` to attempt getting the address of the procedure
- Returns to the delay import load thunk to call the now-loaded entry point

The helper function can call back to a notification hook in your program after each of the following actions:

- When the helper function starts up
- Just before `LoadLibrary` is called in the helper function
- Just before `GetProcAddress` is called in the helper function
- If the call to `LoadLibrary` in the helper function fails
- If the call to `GetProcAddress` in the helper function fails
- After the helper function is done processing

Each of these hook points can return a value that alters normal processing of the helper routine in some manner, except the return to the delay import load thunk.

The default helper code can be found in *`delayhlp.cpp`* and *`delayimp.h`* in the MSVC *`include`* directory. It's compiled into *`delayimp.lib`* in the MSVC *`lib`* directory for your target architecture. You'll need to include this library in your compilations unless you write your own helper function.

Delay load helper calling conventions, parameters, and return type

The prototype for the delay load helper routine is:

```
C

FARPROC WINAPI __delayLoadHelper2(
    PCImgDelayDescr pidd,
    FARPROC * ppfnIATEntry
);
```

Parameters

pidd

A **const** pointer to a ImgDelayDescr that contains the offsets of various import-related data, a timestamp for binding information, and a set of attributes that provide further information about the descriptor content. Currently there's only one attribute, dlattrRva, which indicates that the addresses in the descriptor are relative virtual addresses. For more information, see the declarations in *delayimp.h*.

The pointers in the delay descriptor (ImgDelayDescr in *delayimp.h*) use relative virtual addresses (RVAs) to work as expected in both 32-bit and 64-bit programs. To use them, convert these RVAs back to pointers by using the function PFromRva, found in *delayhlp.cpp*. You can use this function on each of the fields in the descriptor to convert them back to either 32-bit or 64-bit pointers. The default delay load helper function is a good template to use as an example.

For the definition of the PCImgDelayDescr structure, see Structure and constant definitions.

ppfnIATEntry

A pointer to a slot in the delay load import address table (IAT). It's the slot that's updated with the address of the imported function. The helper routine needs to store the same value that it returns into this location.

Expected return values

If the helper function is successful, it returns the address of the imported function.

If the function fails, it raises a structured exception and returns 0. Three types of exceptions can be raised:

- Invalid parameter, which happens if the attributes in `pidd` aren't specified correctly. Treat this as an unrecoverable error.
- `LoadLibrary` failed on the specified DLL.
- Failure of `GetProcAddress`.

It's your responsibility to handle these exceptions. For more information, see Error handling and notification.

Remarks

The calling convention for the helper function is `__stdcall`. The type of the return value isn't relevant, so `FARPROC` is used. This function has C linkage, which means it needs to be wrapped by `extern "C"` when declared in C++ code. The `ExternC` macro takes care of this wrapper for you.

To use your helper routine as a notification hook, your code must specify the appropriate function pointer to return. The thunk code the linker generates then takes that return value as the real target of the import and jumps directly to it. If you don't want to use your helper routine as a notification hook, store the return value of the helper function in `ppfnIATEntry`, the passed-in function pointer location.

Sample hook function

The following code shows how to implement a basic hook function.

C

```
FARPROC WINAPI delayHook(unsigned dliNotify, PDelayLoadInfo pdli)
{
    switch (dliNotify) {
        case dliStartProcessing :

            // If you want to return control to the helper, return 0.
            // Otherwise, return a pointer to a FARPROC helper function
            // that will be used instead, thereby bypassing the rest
            // of the helper.

            break;

        case dliNotePreLoadLibrary :
```

```
            // If you want to return control to the helper, return 0.
            // Otherwise, return your own HMODULE to be used by the
            // helper instead of having it call LoadLibrary itself.

            break;

        case dliNotePreGetProcAddress :

            // If you want to return control to the helper, return 0.
            // If you choose you may supply your own FARPROC function
            // address and bypass the helper's call to GetProcAddress.

            break;

        case dliFailLoadLib :

            // LoadLibrary failed.
            // If you don't want to handle this failure yourself, return 0.
            // In this case the helper will raise an exception
            // (ERROR_MOD_NOT_FOUND) and exit.
            // If you want to handle the failure by loading an alternate
            // DLL (for example), then return the HMODULE for
            // the alternate DLL. The helper will continue execution with
            // this alternate DLL and attempt to find the
            // requested entrypoint via GetProcAddress.

            break;

        case dliFailGetProc :

            // GetProcAddress failed.
            // If you don't want to handle this failure yourself, return 0.
            // In this case the helper will raise an exception
            // (ERROR_PROC_NOT_FOUND) and exit.
            // If you choose, you may handle the failure by returning
            // an alternate FARPROC function address.

            break;

        case dliNoteEndProcessing :

            // This notification is called after all processing is done.
            // There is no opportunity for modifying the helper's behavior
            // at this point except by longjmp()/throw()/RaiseException.
            // No return value is processed.

            break;

        default :

            return NULL;
    }

    return NULL;
```

```
}

/*
and then at global scope somewhere:

ExternC const PfnDliHook __pfnDliNotifyHook2 = delayHook;
ExternC const PfnDliHook __pfnDliFailureHook2 = delayHook;
*/
```

Delay load structure and constant definitions

The default delay load helper routine uses several structures to communicate with the hook functions and during any exceptions. These structures are defined in `delayimp.h`. Here are the macros, typedefs, notification and failure values, information structures, and the pointer-to-hook-function type passed to the hooks:

C

```
#define _DELAY_IMP_VER  2

#if defined(__cplusplus)
#define ExternC extern "C"
#else
#define ExternC extern
#endif

typedef IMAGE_THUNK_DATA *          PImgThunkData;
typedef const IMAGE_THUNK_DATA *    PCImgThunkData;
typedef DWORD                       RVA;

typedef struct ImgDelayDescr {
    DWORD           grAttrs;        // attributes
    RVA             rvaDLLName;     // RVA to dll name
    RVA             rvaHmod;        // RVA of module handle
    RVA             rvaIAT;         // RVA of the IAT
    RVA             rvaINT;         // RVA of the INT
    RVA             rvaBoundIAT;    // RVA of the optional bound IAT
    RVA             rvaUnloadIAT;   // RVA of optional copy of original IAT
    DWORD           dwTimeStamp;    // 0 if not bound,
                                    // O.W. date/time stamp of DLL bound to
(Old BIND)
    } ImgDelayDescr, * PImgDelayDescr;

typedef const ImgDelayDescr *   PCImgDelayDescr;

enum DLAttr {                   // Delay Load Attributes
    dlattrRva = 0x1,                // RVAs are used instead of pointers
                                    // Having this set indicates a VC7.0
                                    // and above delay load descriptor.

    };
```

```
//
// Delay load import hook notifications
//
enum {
    dliStartProcessing,             // used to bypass or note helper only
    dliNoteStartProcessing = dliStartProcessing,

    dliNotePreLoadLibrary,          // called just before LoadLibrary, can
                                    //  override w/ new HMODULE return val
    dliNotePreGetProcAddress,       // called just before GetProcAddress,
can
                                    //  override w/ new FARPROC return value
    dliFailLoadLib,                 // failed to load library, fix it by
                                    //  returning a valid HMODULE
    dliFailGetProc,                 // failed to get proc address, fix it by
                                    //  returning a valid FARPROC
    dliNoteEndProcessing,           // called after all processing is done,
no
                                    //  bypass possible at this point except
                                    //  by longjmp()/throw()/RaiseException.
    };

typedef struct DelayLoadProc {
    BOOL                fImportByName;
    union {
        LPCSTR          szProcName;
        DWORD           dwOrdinal;
        };
    } DelayLoadProc;

typedef struct DelayLoadInfo {
    DWORD               cb;         // size of structure
    PCImgDelayDescr     pidd;       // raw form of data (everything is
there)
    FARPROC *           ppfn;       // points to address of function to load
    LPCSTR              szDll;      // name of dll
    DelayLoadProc       dlp;        // name or ordinal of procedure
    HMODULE             hmodCur;    // the hInstance of the library we have
loaded
    FARPROC             pfnCur;     // the actual function that will be
called
    DWORD               dwLastError;// error received (if an error
notification)
    } DelayLoadInfo, * PDelayLoadInfo;

typedef FARPROC (WINAPI *PfnDliHook)(
    unsigned        dliNotify,
    PDelayLoadInfo  pdli
    );
```

Calculate necessary values for delay loading

The delay load helper routine needs to calculate two critical pieces of information. To help, there are two inline functions in *delayhlp.cpp* to calculate this information.

- The first, `IndexFromPImgThunkData`, calculates the index of the current import into the three different tables (import address table (IAT), bound import address table (BIAT), and unbound import address table (UIAT)).
- The second, `CountOfImports`, counts the number of imports in a valid IAT.

C

```
// utility function for calculating the index of the current import
// for all the tables (INT, BIAT, UIAT, and IAT).
__inline unsigned
IndexFromPImgThunkData(PCImgThunkData pitdCur, PCImgThunkData pitdBase) {
    return pitdCur - pitdBase;
    }

// utility function for calculating the count of imports given the base
// of the IAT. NB: this only works on a valid IAT!
__inline unsigned
CountOfImports(PCImgThunkData pitdBase) {
    unsigned        cRet = 0;
    PCImgThunkData  pitd = pitdBase;
    while (pitd->u1.Function) {
        pitd++;
        cRet++;
        }
    return cRet;
    }
```

Support unload of a delay-loaded DLL

When a delay-loaded DLL gets loaded, the default delay-load helper checks to see if the delay-load descriptors have a pointer and a copy of the original import address table (IAT) in the `pUnloadIAT` field. If so, the helper saves a pointer in a list to the import delay descriptor. This entry lets the helper function find the DLL by name, to support unloading that DLL explicitly.

Here are the associated structures and functions for explicitly unloading a delay-loaded DLL:

C++

```
//
// Unload support from delayimp.h
//
```

```
// routine definition; takes a pointer to a name to unload

ExternC
BOOL WINAPI
__FUnloadDelayLoadedDLL2(LPCSTR szDll);

// structure definitions for the list of unload records
typedef struct UnloadInfo * PUnloadInfo;
typedef struct UnloadInfo {
    PUnloadInfo     puiNext;
    PCImgDelayDescr pidd;
    } UnloadInfo;

// from delayhlp.cpp
// the default delay load helper places the unloadinfo records in the
// list headed by the following pointer.
ExternC
PUnloadInfo __puiHead;
```

The `UnloadInfo` structure is implemented using a C++ class that uses `LocalAlloc` and `LocalFree` implementations as its `operator new` and `operator delete`, respectively. These options are kept in a standard linked list that uses `__puiHead` as the head of the list.

When you call `__FUnloadDelayLoadedDLL`, it attempts to find the name you provide in the list of loaded DLLs. (An exact match is required.) If found, the copy of the IAT in `pUnloadIAT` is copied over the top of the running IAT to restore the thunk pointers. Then, the library is freed by using `FreeLibrary`, the matching `UnloadInfo` record is unlinked from the list and deleted, and `TRUE` is returned.

The argument to the function `__FUnloadDelayLoadedDLL2` is case-sensitive. For example, you would specify:

C++

```
__FUnloadDelayLoadedDLL2("user32.dll");
```

and not:

C++

```
__FUnloadDelayLoadedDLL2("User32.DLL");
```

For an example of unloading a delay-loaded DLL, see Explicitly unload a delay-loaded DLL.

Develop your own delay load helper function

You may want to provide your own version of the delay load helper routine. In your own routine, you can do specific processing based on the names of the DLL or imports. There are two ways to insert your own code: Code your own helper function, possibly based on the supplied code. Or, hook the supplied helper to call your own function by using the notification hooks.

Code your own helper

Creating your own helper routine is straightforward. You can use the existing code as a guide for your new function. Your function must use the same calling conventions as the existing helper. And, if it returns to the linker-generated thunks, it must return a proper function pointer. Once you've created your code, you may either satisfy the call or get out of the call, however you like.

Use the start processing notification hook

It's probably easiest to provide a new pointer to a user-supplied notification hook function that takes the same values as the default helper for the `dliStartProcessing` notification. At that point, the hook function can essentially become the new helper function, because a successful return to the default helper bypasses all further processing in the default helper.

See also

Linker support for delay-loaded DLLs

Additional MSVC Build Tools

Article • 08/03/2021

Visual Studio provides the following command-line utilities for viewing or manipulating build output:

- LIB.EXE is used to create and manage a library of Common Object File Format (COFF) object files. It can also be used to create export files and import libraries to reference exported definitions.
- EDITBIN.EXE is used to modify COFF binary files.
- DUMPBIN.EXE displays information (such as a symbol table) about COFF binary files.
- NMAKE reads and executes makefiles.
- ERRLOOK, the Error Lookup utility, retrieves a system error message or module error message based on the value entered.
- XDCMake. A tool for processing source code files that contain documentation comments marked up with XML tags.
- BSCMAKE.EXE (provided for backward compatibility only) builds a browse information file (.bsc) that contains information about the symbols (classes, functions, data, macros, and types) in your program. You view this information in browse windows within the development environment. (A .bsc file can also be built in the development environment.)

The Windows SDK also has several build tools, including RC.EXE, which the C++ compiler invokes to compile native Windows resources such as dialogs, property pages, bitmaps, string tables and so on.

See also

C/C++ Building Reference
Decorated Names
MSVC Compiler Options
MSVC Linker Options

NMAKE Reference

Article • 09/30/2021

The Microsoft Program Maintenance Utility (NMAKE.EXE) is a command-line tool included with Visual Studio. It builds projects based on commands that are contained in a description file, usually called a *makefile*.

NMAKE must run in a Developer Command Prompt window. A Developer Command Prompt window has the environment variables set for the tools, libraries, and include file paths required to build at the command line. For details on how to open a Developer Command Prompt window, see Use the MSVC toolset from the command line.

What do you want to know more about?

Running NMAKE

Makefile contents and features

Sample Makefile

Description blocks

Commands in a Makefile

Macros and NMAKE

Inference rules

Dot directives

Makefile preprocessing

See also

Use the MSVC toolset from the command line
Additional MSVC Build Tools
Visual Studio Projects - C++
Debugging in Visual Studio
C/C++ Building Reference

Create a C++ makefile project

Article • 03/03/2022

A *makefile* is a text file that contains instructions for how to compile and link (or *build*) a set of source code files. A program (often called a *make* program) reads the makefile and invokes a compiler, linker, and possibly other programs to make an executable file. The Microsoft program is called NMAKE.

If you have an existing makefile project, you have these choices if you want to edit, build, and debug in the Visual Studio IDE:

- Create a makefile project in Visual Studio that uses your existing makefile to configure a .vcxproj file that Visual Studio will use for IntelliSense. (You won't have all the IDE features that you get with a native MSBuild project.) See To create a makefile project below.
- Use the **Create New Project from Existing Code Files** wizard to create a native MSBuild project from your source code. The original makefile won't be used anymore. For more information, see How to: Create a C++ Project from Existing Code.
- **Visual Studio 2017 and later**: Use the **Open Folder** feature to edit and build a makefile project as-is without any involvement of the MSBuild system. For more information, see Open Folder projects for C++.
- **Visual Studio 2019 and later**: Create a UNIX makefile project for Linux.

To create a makefile project with the makefile project template

In Visual Studio 2017 and later, the Makefile project template is available when the C++ Desktop Development workload is installed.

Follow the wizard to specify the commands and environment used by your makefile. You can then use this project to build your code in Visual Studio.

By default, the makefile project displays no files in Solution Explorer. The makefile project specifies the build settings, which are reflected in the project's property page.

The output file that you specify in the project has no effect on the name that the build script generates. It declares only an intention. Your makefile still controls the build process and specifies the build targets.

To create a makefile project in Visual Studio

1. From the Visual Studio main menu, choose **File** > **New** > **Project** and type "makefile" into the search box. If you see more than one project template, select from the options depending on your target platform.

2. **Windows only**: In the Makefile project **Debug Configuration Settings** page, provide the command, output, clean, and rebuild information for debug and retail builds. Choose **Next** if you want to specify different settings for a Release configuration.

3. Choose **Finish** to close the dialog and open the newly created project in **Solution Explorer**.

You can view and edit the project's properties in its property page. For more information about displaying the property page, see Set C++ compiler and build properties in Visual Studio.

Makefile project wizard

After you create a makefile project, you can view and edit each of the following options in the **Nmake** page of the project's property page.

- **Build command line:** Specifies the command line to run when the user selects Build from the Build menu. Displayed in the Build command line field on the Nmake page of the project's property page.

- **Output:** Specifies the name of the file that will contain the output for the command line. By default, this option is based on the project name. Displayed in the Output field on the Nmake page of the project's property page.

- **Clean commands:** Specifies the command line to run when the user selects Clean from the Build menu. Displayed in the Clean command line field on the Nmake page of the project's property page.

- **Rebuild command line:** Specifies the command line to run when the user selects Rebuild from the Build menu. Displayed in the Rebuild all command line field on the Nmake page of the project's property page.

How to: Enable IntelliSense for Makefile Projects

IntelliSense fails in makefile projects when certain project settings or compiler options are set up incorrectly. Follow these steps to configure makefile projects so that IntelliSense works as expected:

1. Open the **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.

2. Select the **Configuration Properties** > **NMake** property page.

3. Modify properties under **IntelliSense** as appropriate:

 - Set the **Preprocessor Definitions** property to define any preprocessor symbols in your makefile project. For more information, see /D (Preprocessor Definitions).

 - Set the **Include Search Path** property to specify the list of directories that the compiler will search to resolve file references that are passed to preprocessor directives in your makefile project. For more information, see /I (Additional Include Directories).

 - For projects that are built using CL.EXE from a Command Window, set the **INCLUDE** environment variable to specify directories that the compiler will search to resolve file references that are passed to preprocessor directives in your makefile project.

 - Set the **Forced Includes** property to specify which header files to process when building your makefile project. For more information, see /FI (Name Forced Include File).

 - Set the **Assembly Search Path** property to specify the list of directories that the compiler will search to resolve references to .NET assemblies in your project. For more information, see /AI (Specify Metadata Directories).

 - Set the **Forced Using Assemblies** property to specify which .NET assemblies to process when building your makefile project. For more information, see /FU (Name Forced #using File).

 - Set the **Additional Options** property to specify other compiler switches to be used by IntelliSense when parsing C++ files.

4. Choose **OK** to close the property pages.

5. Use the **Save All** command to save the modified project settings.

The next time you open your makefile project in the Visual Studio development environment, run the **Clean Solution** command and then the **Build Solution** command on your makefile project. IntelliSense should work properly in the IDE.

See also

Using IntelliSense
NMAKE Reference
How to: Create a C++ project from existing code
Special characters in a makefile
Makefile contents and features

Running NMAKE

Article • 02/17/2022

Syntax

> **`NMAKE`** [*option* ...] [*macros* ...] [*targets* ...] [**`@`***command-file* ...]

Remarks

NMAKE builds only specified *targets* or, when none is specified, the first target in the makefile. The first makefile target can be a pseudotarget that builds other targets. NMAKE uses makefiles specified with **`/F`**, or if **`/F`** isn't specified, the Makefile file in the current directory. If no makefile is specified, it uses inference rules to build command-line *targets*.

The *command-file* text file (or response file) contains command-line input. Other input can precede or follow @*command-file*. A path is permitted. In *command-file*, line breaks are treated as spaces. Enclose macro definitions in quotation marks if they contain spaces.

NMAKE options

NMAKE options are described in the following table. Options are preceded by either a slash (**`/`**) or a dash (**`-`**), and aren't case-sensitive. Use !CMDSWITCHES to change option settings in a makefile or in *`Tools.ini`*.

Option	Purpose
`/A`	Forces build of all evaluated targets, even if not out-of-date compared to dependents. Doesn't force builds of unrelated targets.
`/B`	Forces build even if timestamps are equal. Recommended only for fast systems (resolution of two seconds or less).
`/C`	Suppresses default output, including nonfatal NMAKE errors or warnings, timestamps, and NMAKE copyright message. Suppresses warnings issued by **`/K`**.
`/D`	Displays timestamps of each evaluated target and dependent and a message when a target doesn't exist. Useful with **`/P`** for debugging a makefile. Use **`!CMDSWITCHES`** to set or clear **`/D`** for part of a makefile.

Option	Purpose
`/E`	Causes environment variables to override makefile macro definitions.
`/ERRORREPORT` [`NONE` \| `PROMPT` \| `QUEUE` \| `SEND`]	Deprecated. Windows Error Reporting (WER) settings control reporting.
`/F` *filename*	Specifies *filename* as a makefile. Spaces or tabs can precede *filename*. Specify `/F` once for each makefile. To supply a makefile from standard input, specify a dash (`-`) for *filename*, and end keyboard input with either **F6** or **CTRL+Z**.
`/G`	Displays the makefiles included with the `!INCLUDE` directive. For more information, see Makefile preprocessing directives.
`/HELP`, `/?`	Displays a brief summary of NMAKE command-line syntax.
`/I`	Ignores exit codes from all commands. To set or clear `/I` for part of a makefile, use `!CMDSWITCHES`. To ignore exit codes for part of a makefile, use a dash (`-`) command modifier or .IGNORE. Overrides `/K` if both are specified.
`/K`	Continues building unrelated dependencies, if a command returns an error. Also issues a warning and returns an exit code of 1. By default, NMAKE halts if any command returns a nonzero exit code. Warnings from `/K` are suppressed by `/C`; `/I` overrides `/K` if both are specified.
`/N`	Displays but doesn't execute commands; preprocessing commands are executed. Doesn't display commands in recursive NMAKE calls. Useful for debugging makefiles and checking timestamps. To set or clear `/N` for part of a makefile, use `!CMDSWITCHES`.
`/NOLOGO`	Suppresses the NMAKE copyright message.
`/P`	Displays information (macro definitions, inference rules, targets, .SUFFIXES list) to standard output, and then runs the build. If no makefile or command-line target exists, it displays information only. Use with `/D` to debug a makefile.
`/Q`	Checks timestamps of targets; doesn't run the build. Returns a zero exit code if all targets are up to date, and a nonzero exit code if any target is out of date. Preprocessing commands are executed. Useful when running NMAKE from a batch file.
`/R`	Clears the `.SUFFIXES` list and ignores inference rules and macros that are defined in the `Tools.ini` file or that are predefined.
`/S`	Suppresses display of executed commands. To suppress display in part of a makefile, use the `@` command modifier or .SILENT. To set or clear `/S` for part of a makefile, use `!CMDSWITCHES`.

Option	Purpose
`/T`	Updates timestamps of command-line targets (or first makefile target) and executes preprocessing commands but doesn't run the build.
`/U`	Must be used in conjunction with `/N`. Dumps inline NMAKE files so that the `/N` output can be used as a batch file.
`/X` *filename*	Sends NMAKE error output to *filename* instead of standard error. Spaces or tabs can precede *filename*. To send error output to standard output, specify a dash (`-`) for *filename*. Doesn't affect output from commands to standard error.
`/Y`	Disables batch-mode inference rules. When this option is selected, all batch-mode inference rules are treated as regular inference rules.

Tools.ini and NMAKE

NMAKE reads *`Tools.ini`* before it reads makefiles, unless `/R` is used. It looks for *`Tools.ini`* first in the current directory, and then in the directory specified by the INIT environment variable. The section for NMAKE settings in the initialization file begins with `[NMAKE]` and can contain any makefile information. Specify a comment on a separate line beginning with a number sign (`#`).

Exit Codes from NMAKE

NMAKE returns the following exit codes:

Code	Meaning
0	No error (possibly a warning)
1	Incomplete build (issued only when `/K` is used)
2	Program error, possibly caused by one of these issues: - A syntax error in the makefile - An error or exit code from a command - An interruption by the user
4	System error — out of memory
255	Target isn't up to date (issued only when `/Q` is used)

See also

NMAKE Reference

NMAKE makefile contents and features

Article • 09/30/2021

A makefile contains:

- Description blocks
- Commands
- Macros
- Inference rules
- Dot directives
- Preprocessing directives

For a sample, see Sample makefile.

NMAKE supports other features, such as wildcards, long filenames, comments, and escapes for special characters.

Wildcards and NMAKE

NMAKE expands filename wildcards (`*` and `?`) in dependency lines. A wildcard specified in a command is passed to the command; NMAKE doesn't expand it.

Long filenames in a makefile

Enclose long filenames in double quotation marks, as follows:

makefile

```
all : "VeryLongFileName.exe"
```

Comments in a makefile

Precede a comment with a number sign (`#`). NMAKE ignores text from the number sign to the next newline character.

Examples:

makefile

```
# Comment on line by itself
OPTIONS = /MAP  # Comment on macro definition line

all.exe : one.obj two.obj  # Comment on dependency line
    link one.obj two.obj
# Comment in commands block
#   copy *.obj \objects  # Command turned into comment
    copy one.exe \release

.obj.exe:  # Comment on inference rule line
    link $<

my.exe : my.obj ; link my.obj  # Err: cannot comment this
# Error: # must be the first character
.obj.exe: ; link $<  # Error: cannot comment this
```

To specify a literal number sign, precede it with a caret (`^`), as follows:

makefile

```
DEF = ^#define  #Macro for a C preprocessing directive
```

Special characters in a makefile

To use an NMAKE special character as a literal character, place a caret (`^`) in front of it as an escape. NMAKE ignores carets that precede other characters. The special characters are:

`: ; # ( ) $ ^ \ { } ! @ -`

A caret (`^`) within a quoted string is treated as a literal caret character. A caret at the end of a line inserts a literal newline character in a string or macro.

In macros, a backslash (`\`) followed by a newline character is replaced by a space.

In commands, a percent symbol (`%`) is a file specifier. To represent `%` literally in a command, specify a double percent sign (`%%`) in place of a single one. In other situations, NMAKE interprets a single `%` literally, but it always interprets a double `%%` as a single `%`. Therefore, to represent a literal `%%`, specify either three percent signs, `%%%`, or four percent signs, `%%%%`.

To use the dollar sign (`$`) as a literal character in a command, specify two dollar signs (`$$`). This method can also be used in other situations where `^$` works.

See also

NMAKE Reference

Sample Makefile

Article • 09/30/2021

This topic contains a sample makefile.

Sample

Code

makefile

```
# Sample makefile

!include <win32.mak>

all: simple.exe challeng.exe

.c.obj:
  $(cc) $(cdebug) $(cflags) $(cvars) $*.c

simple.exe: simple.obj
  $(link) $(ldebug) $(conflags) -out:simple.exe simple.obj $(conlibs)
lsapi32.lib

challeng.exe: challeng.obj md4c.obj
  $(link) $(ldebug) $(conflags) -out:challeng.exe $** $(conlibs) lsapi32.lib
```

See also

Makefile contents and features

Description blocks

Article • 09/30/2021

Description blocks form the core of a makefile. They describe the *targets*, or files to create, and their *dependencies*, the files needed to create the targets. A description block may include *commands*, that describe how to create the targets from the dependencies. A description block is a dependency line, optionally followed by a commands block:

makefile

```
targets... : dependents...
    commands...
```

Dependency lines

A *dependency line* specifies one or more targets, and zero or more dependents. If a target doesn't exist, or has an earlier timestamp than a dependent, NMAKE executes the commands in the command block. NMAKE also executes the command block if the target is a pseudotarget. Here's an example dependency line:

makefile

```
hi_bye.exe : hello.obj goodbye.obj helper.lib
```

In this dependency line, `hi_bye.exe` is the target. Its dependencies are `hello.obj`, `goodbye.obj`, and `helper.lib`. The dependency line tells NMAKE to build the target whenever `hello.obj`, `goodbye.obj`, or `helper.lib` has changed more recently than `hi_bye.exe`.

A target must be at the start of the line. It can't be indented with any spaces or tabs. Use a colon (`:`) to separate targets from dependents. Spaces or tabs are allowed between targets, the colon separator (`:`), and dependents. To split the dependency line, use a backslash (`\`) after a target or dependent.

Before it executes command blocks, NMAKE scans all the dependencies and any applicable inference rules to build a *dependency tree*. A dependency tree specifies the steps required to fully update the target. NMAKE checks recursively whether a dependent is itself a target in another dependency list. After it builds the dependency tree, NMAKE checks time stamps. If any dependents in the tree are newer than the target, NMAKE builds the target.

Targets

The targets section of a dependency line specifies one or more targets. A target can be any valid filename, directory name, or pseudotarget. Separate multiple targets by using one or more spaces or tabs. Targets aren't case-sensitive. Paths are permitted with filenames. A target and its path can't exceed 256 characters. If the target preceding the colon is a single character, use a separating space. Otherwise, NMAKE interprets the letter-colon combination as a drive specifier.

Multiple targets

NMAKE evaluates multiple targets in a single dependency as if each were specified in a separate description block.

For example, this rule:

makefile

```
bounce.exe leap.exe : jump.obj
   echo Building...
```

is evaluated as:

makefile

```
bounce.exe : jump.obj
   echo Building...

leap.exe : jump.obj
   echo Building...
```

Cumulative dependencies

Dependencies are cumulative in a description block, if a target is repeated.

For example, this set of rules,

makefile

```
bounce.exe : jump.obj
bounce.exe : up.obj
   echo Building bounce.exe...
```

is evaluated as:

makefile

```
bounce.exe : jump.obj up.obj
   echo Building bounce.exe...
```

When you have multiple targets in multiple dependency lines in a single description block, NMAKE evaluates them as if each were specified in a separate description block. However, only targets in the last dependency line use the commands block. NMAKE attempts to use an inference rule for the other targets.

For example, this set of rules,

makefile

```
leap.exe bounce.exe : jump.obj
bounce.exe climb.exe : up.obj
   echo Building bounce.exe...
```

is evaluated as:

makefile

```
leap.exe : jump.obj
# invokes an inference rule

bounce.exe : jump.obj up.obj
   echo Building bounce.exe...

climb.exe : up.obj
   echo Building bounce.exe...
```

Targets in multiple description blocks

To update a target in more than one description block using different commands, specify two consecutive colons (::) between targets and dependents.

makefile

```
target.lib :: one.asm two.asm three.asm
    ml one.asm two.asm three.asm
    lib target one.obj two.obj three.obj
target.lib :: four.c five.c
    cl /c four.c five.c
    lib target four.obj five.obj
```

Dependency side effects

You might specify a target with a colon (:) in two dependency lines in different locations. If commands appear after only one of the lines, NMAKE interprets the dependencies as if the lines were adjacent or combined. It doesn't invoke an inference rule for the dependency that has no commands. Instead, NMAKE assumes the dependencies belong to one description block, and executes the commands specified with the other dependency. Consider this set of rules:

makefile

```
bounce.exe : jump.obj
   echo Building bounce.exe...

bounce.exe : up.obj
```

is evaluated as:

makefile

```
bounce.exe : jump.obj up.obj
   echo Building bounce.exe...
```

This effect doesn't occur if a double colon (`::`) is used. For example, this set of rules:

makefile

```
bounce.exe :: jump.obj
   echo Building bounce.exe...

bounce.exe :: up.obj
```

is evaluated as:

makefile

```
bounce.exe : jump.obj
   echo Building bounce.exe...

bounce.exe : up.obj
# invokes an inference rule
```

Pseudotargets

A *pseudotarget* is a label used in place of a filename in a dependency line. It's interpreted as a file that doesn't exist, and so is out-of-date. NMAKE assumes a pseudotarget's timestamp is the same as the most recent of all its dependents. If it has no dependents, the current time is assumed. If a pseudotarget is used as a target, its commands are always executed. A pseudotarget used as a dependent must also appear as a target in another dependency. However, that dependency doesn't need to have a commands block.

Pseudotarget names follow the filename syntax rules for targets. However, if the name doesn't have an extension, it can exceed the 8-character limit for filenames, and can be up to 256 characters long.

Pseudotargets are useful when you want NMAKE to build more than one target automatically. NMAKE only builds targets specified on the command line. Or, if no command-line target is specified, it builds only the first target in the first dependency in the makefile. You can tell NMAKE to build multiple targets without listing them individually on the command line. Write a description block with a dependency containing a pseudotarget, and list the targets you want to build as its dependents. Then, place this description block first in the makefile, or specify the pseudotarget on the NMAKE command line.

In this example, UPDATE is a pseudotarget.

makefile

```
UPDATE : *.*
!COPY $** c:\product\release
```

When UPDATE is evaluated, NMAKE copies all files in the current directory to the specified drive and directory.

In the following makefile, the pseudotarget `all` builds both `project1.exe` and `project2.exe` if either `all` or no target is specified on the command line. The pseudotarget `setenv` changes the LIB environment variable before the `.exe` files are updated:

makefile

```
all : setenv project1.exe project2.exe

project1.exe : project1.obj
    LINK project1;

project2.exe : project2.obj
    LINK project2;
```

```
setenv :
    set LIB=\project\lib
```

Dependents

In a dependency line, specify zero or more dependents after the colon (`:`) or double colon (`::`), using any valid filename or pseudotarget. Separate multiple dependents by using one or more spaces or tabs. Dependents aren't case-sensitive. Paths are permitted with filenames.

Inferred dependents

Along with dependents you explicitly list in the dependency line, NMAKE can assume an *inferred dependent*. An inferred dependent is derived from an inference rule, and is evaluated before explicit dependents. When an inferred dependent is out-of-date compared to its target, NMAKE invokes the command block for the dependency. If an inferred dependent doesn't exist, or is out-of-date compared to its own dependents, NMAKE first updates the inferred dependent. For more information about inferred dependents, see Inference rules.

Search paths for dependents

You can specify an optional search path for each dependent. Here's the syntax to specify a set of directories to search:

> {*directory*[;*directory*...]}*dependent*

Enclose directory names in braces (`{ }`). Separate multiple directories with a semicolon (`;`). No spaces or tabs are allowed. NMAKE looks for the dependent first in the current directory, and then in the list of directories in the order specified. You can use a macro to specify part or all of a search path. Only the specified dependent uses this search path.

Directory search path example

This dependency line shows how to create a directory specification for a search:

```
makefile
```

```
reverse.exe : {\src\omega;e:\repo\backwards}retro.obj
```

The target `reverse.exe` has one dependent, `retro.obj`. The brace-enclosed list specifies two directories. NMAKE searches for `retro.obj` in the current directory first. If it isn't there, NMAKE searches the `\src\omega` directory, then the `e:\repo\backwards` directory.

See also

NMAKE Reference

Commands in a makefile

Article • 09/30/2021

A description block or inference rule specifies a block of commands to run if the dependency is out-of-date. NMAKE displays each command before running it, unless `/S`, `.SILENT`, `!CMDSWITCHES`, or `@` is used. NMAKE looks for a matching inference rule if a description block isn't followed by a commands block.

A commands block contains one or more commands, each on its own line. No blank line can appear between the dependency or rule and the commands block. However, a line containing only spaces or tabs can appear; this line is interpreted as a null command, and no error occurs. Blank lines are permitted between command lines.

A command line begins with one or more spaces or tabs. A backslash (`\`) followed by a newline character is interpreted as a space in the command. Use a backslash at the end of a line to continue a command onto the next line. NMAKE interprets the backslash literally if any other character, including a space or tab, follows the backslash.

A command preceded by a semicolon (`;`) can appear on a dependency line or inference rule, whether a commands block follows or not:

makefile

```
project.obj : project.c project.h ; cl /c project.c
```

Command modifiers

You can specify one or more command modifiers preceding a command, optionally separated by spaces or tabs. As with commands, modifiers must be indented.

Modifier	Purpose
`@`*command*	Prevents display of the command. Display by commands is not suppressed. By default, NMAKE echoes all executed commands. Use `/S` to suppress display for the entire makefile; use `.SILENT` to suppress display for part of the makefile.
`-`[*number*] *command*	Turns off error checking for *command*. By default, NMAKE halts when a command returns a nonzero exit code. If *-number* is used, NMAKE stops if the exit code exceeds *number*. Spaces or tabs can't appear between the dash and *number*. At least one space or tab must appear between *number* and *command*. Use `/I` to turn off error checking for the entire makefile; use `.IGNORE` to turn off error checking for part of the makefile.

Modifier	Purpose
`!` *command*	Executes *command* for each dependent file if *command* uses `$**` (all dependent files in the dependency) or `$?` (all dependent files in the dependency with a later timestamp than the target).

Filename-parts syntax

Filename-parts syntax in commands represents components of the first dependent filename (which may be an implied dependent). Filename components are the file's drive, path, base name, and extension as specified, not as it exists on disk. Use `%s` to represent the complete filename. Use `%|`[*parts*]`F` (a vertical bar character follows the percent symbol) to represent parts of the filename, where *parts* can be zero or more of the following letters, in any order.

Letter	Description
No letter	Complete name (same as `%s`)
`d`	Drive
`p`	Path
`f`	File base name
`e`	File extension

For example, if the filename is `c:\prog.exe`:

- `%s` becomes `c:\prog.exe`
- `%|F` becomes `c:\prog.exe`
- `%|dF` becomes `c`
- `%|pF` becomes `c:\`
- `%|fF` becomes `prog`
- `%|eF` becomes `exe`

What do you want to know more about?

Inline files in a makefile

See also

NMAKE Reference

Inline files in a makefile

Article • 09/30/2021

An inline file contains text you specify in the makefile. Its name can be used in commands as input (for example, a LINK command file), or it can pass commands to the operating system. The file is created on disk when a command that creates the file is run.

Specify an inline file

Specify two angle brackets (`<<`) in the command where *filename* is to appear. The angle brackets can't be a macro expansion. The *filename* is optional:

```
makefile

<<filename
```

When the command is run, the angle brackets are replaced by *filename*, if specified, or by a unique NMAKE-generated name. If specified, *filename* must follow angle brackets without a space or tab. A path is permitted. No extension is required or assumed. If *filename* is specified, the file is created in the current or specified directory, overwriting any existing file by that name. Otherwise, it's created in the `TMP` directory (or the current directory, if the `TMP` environment variable isn't defined). If a previous *filename* is reused, NMAKE replaces the previous file.

Create inline file text

Inline files are temporary or permanent.

```
makefile

inline_text
.
.
.
<<[KEEP | NOKEEP]
```

Specify your *inline_text* on the first line after the command. Mark the end with double angle brackets (`<<`) at the beginning of a separate line, followed by an optional **`KEEP`** or **`NOKEEP`**. The file contains all *inline_text* before the delimiting brackets. The *inline_text* can

have macro expansions and substitutions, but not directives or makefile comments. Spaces, tabs, and newline characters are treated literally.

A temporary file exists for the duration of the session and can be reused by other commands. Specify `KEEP` after the closing angle brackets to retain the file after the NMAKE session; an unnamed file is preserved on disk with the generated filename. Specify `NOKEEP` or nothing for a temporary file. `KEEP` and `NOKEEP` are not case sensitive.

Reuse inline files

To reuse an inline file, specify `<<filename` where the file is defined and first used, then reuse *filename* without `<<` later in the same or another command. The command to create the inline file must run before all commands that use the file.

Multiple inline files

A command can create more than one inline file:

makefile

```
command << <<
inline_text
<<[KEEP | NOKEEP]
inline_text
. . .
inline_text
<<[KEEP | NOKEEP]
```

For each file, specify one or more lines of inline text followed by a closing line containing the delimiter and optional `KEEP` or `NOKEEP`. Begin the second file's text on the line following the delimiting line for the first file.

See also

Commands in a Makefile

Macros and NMAKE

Article • 08/03/2021

Macros replace a particular string in the makefile with another string. Using macros, you can:

- Create a makefile that can build different projects.
- Specify options for commands.
- Set environment variables.

You can define your own macros or use NMAKE's predefined macros.

What do you want to know more about?

Defining an NMAKE macro

Using an NMAKE macro

Special NMAKE macros

See also

NMAKE Reference

Define an NMAKE macro

Article • 09/30/2021

An NMAKE macro is defined by using this syntax:

```makefile
macro_name=string
```

The *macro_name* is a case-sensitive combination of letters, digits, and underscores (`_`) up to 1,024 characters long. The *macro_name* can contain an invoked macro. If *macro_name* consists entirely of an invoked macro, the macro being invoked can't be null or undefined.

The *string* can be any sequence of zero or more characters. A *null* string contains zero characters or only spaces or tabs. The *string* can contain a macro invocation.

Special characters in macros

A number sign (`#`) after a definition specifies a comment. To specify a literal number sign in a macro, use a caret (`^`) to escape it, as in `^#`.

A dollar sign (`$`) specifies a macro invocation. To specify a literal `$`, use `$$`.

To extend a definition to a new line, end the line with a backslash (`\`). When the macro is invoked, the backslash and following newline character is replaced with a space. To specify a literal backslash at the end of the line, precede it with a caret (`^`) escape, or follow it with a comment specifier (`#`).

To specify a literal newline character, end the line with a caret (`^`) escape, as in this example:

```makefile
CMDS = cls^
dir
```

Null and undefined macros

Both null and undefined macros expand to null strings, but a macro defined as a null string is considered defined in preprocessing expressions. To define a macro as a null string, specify no characters except spaces or tabs after the equal sign (`=`) in a command line or command file, and enclose the null string or definition in double quotation marks (`" "`). To undefine a macro, use `!UNDEF`. For more information, see Makefile preprocessing directives.

Where to define macros

Define macros in a command line, command file, makefile, or the `Tools.ini` file.

In a makefile or the `Tools.ini` file, each macro definition must appear on a separate line and can't start with a space or tab. Spaces or tabs around the equal sign are ignored. All *string* characters are literal, including surrounding quotation marks and embedded spaces.

In a command line or command file, spaces and tabs delimit arguments and can't surround the equal sign. If *string* has embedded spaces or tabs, enclose either the string itself or the entire macro in double quotation marks (`" "`).

Precedence in macro definitions

If a macro has multiple definitions, NMAKE uses the highest-precedence definition. The following list shows the order of precedence, from highest to lowest:

1. A macro defined on the command line
2. A macro defined in a makefile or include file
3. An inherited environment-variable macro
4. A macro defined in the `Tools.ini` file
5. A predefined macro, such as CC and AS

Use `/E` to cause macros inherited from environment variables to override makefile macros with the same name. Use `!UNDEF` to override a command line.

See also

Macros and NMAKE

Use an NMAKE macro

Article • 02/25/2022

To use a macro, enclose its name in parentheses preceded by a dollar sign (`$`) as follows:

```
makefile

$(macro_name)
```

No spaces are allowed. The parentheses are optional if *macro_name* is a single character. The definition string replaces `$(macro_name)`; an undefined macro is replaced by a null string.

Macro substitution

When *macro_name* is invoked, each occurrence of *string1* in its definition string is replaced by *string2*.

```
makefile

$(macro_name:string1=string2)
```

Macro substitution is case-sensitive and is literal; *string1* and *string2* can't invoke macros. Substitution doesn't modify the original definition. You can replace text in any predefined macro except $$@.

No spaces or tabs precede the colon (`:`); any spaces or tabs after the colon are interpreted as literal. If *string2* is null, all occurrences of *string1* are deleted from the macro's definition string.

Macro functions

NMAKE provides a set of functions that can be used to modify strings, lists of items and file paths. These functions are available in NMAKE starting in Visual Studio 2022.

Function syntax

Functions use the following syntax:

makefile

```
$(function_name arg0,arg1,arg2...)
```

Arguments to a function can be any string and may include nested macro invocations. Except in special cases, arguments can't be null.

Any extra whitespace between the function name and the argument list is ignored. If the first argument requires leading whitespace, then use a macro that contains the needed whitespace:

makefile

```
SINGLESPACE=$(subst ',,' ') # Use "subst" since a normal assignment trims
trailing whitespace.
$(subst $(SINGLESPACE)an,irec,red ant) # Evaluates to "redirect"
```

Commas within an argument list are always considered argument separators and can't be escaped. If any argument requires a literal comma, use a macro that contains a comma instead:

makefile

```
COMMA=,
INPUT=a, b
$(subst $(COMMA) , and ,$(INPUT)) # Evaluates to "a and b"
```

List syntax

Some functions support a whitespace-separated list of items. Extra whitespace is ignored at the beginning of the list, the end of the list, or between each item. Lists produced by a function use a single space between each item as a separator, and don't have leading or trailing whitespace.

For example, the simplest list function is `strip`, which takes a single list argument and produces a list with the exact same items (but with the whitespace cleaned as above):

makefile

```
$(strip a   b   c d    ) # Evaluates to "a b c d"
```

Pattern syntax

Some functions support using a *pattern*. A pattern is a string that contains a single wildcard that can match any number of characters. The first `%` in a pattern is the wildcard, and any later `%` characters are treated as literals. A `%` anywhere before the actual wildcard can be escaped by using `\` (that is, `\%` is treated as a literal `%`). Any `\` that would escape the wildcard can be escaped with another `\` (so `\\%` is treated as a literal `\` followed by the wildcard). To be considered a match, all of the input characters must be matched by the pattern; partial matches aren't supported.

Patterns can be demonstrated using the `filter` function, which only keeps items that match the pattern:

makefile

```
$(filter abc,abc) # Evaluates to "abc" - exactly matches
$(filter bc,abc) # Evaluates to "" - pattern "bc" only matches part of the
item "abc"
$(filter %ef,abcdef) # Evaluates to "abcdef" - wildcard matches "abcd"
$(filter a%f,abcdef) # Evaluates to "abcdef" - wildcard matches "bcde"
$(filter %abc,abc) # Evaluates to "abc" - wildcard doesn't need to match any
characters
$(filter a%c%d,abcd abc%d) # Evaluates to "abc%d" - only the first `%` is a
wildcard, the rest are literals
$(filter a\%b%d,a%bcd) # Evaluates to "a%bcd" - `%` before the wildcard must
be escaped with `\`
$(filter a\\%cd,a\bcd) # Evaluates to "a\bcd" - a `\` that would escape the
wildcard must be escaped with another `\`
$(filter a%c\\%d,abc\\%d) # Evaluates to "abc\\%d" - any `\` after the
wildcard isn't treated as an escape
$(filter \\a%f,\\abcdef) # Evaluates to "\\abcdef" - any `\\` that isn't
directly before the wildcard isn't treated as an escape
```

Functions by category

Function	Purpose	Supported
Text functions	Purpose	Supported
findstring, findstringi	Checks if the input contains a string.	VS 2022 17.0
lowercase	Converts a string to lowercase.	VS 2022 17.2
subst, substi	Replaces all instances of one string with another.	VS 2022 17.0

Function	Purpose	Supported
uppercase	Converts a string to uppercase.	VS 2022 17.2
List functions	**Purpose**	**Supported**
filter, filteri	Keeps items in a list that match at least one pattern.	VS 2022 17.0
filterout, filterouti	Keeps items in a list that don't match any patterns.	VS 2022 17.0
patsubst, patsubsti	Transforms each item that matches a pattern, items that don't match are left as-is.	VS 2022 17.1
strip	Cleans the whitespace in and around a list of items.	VS 2022 17.0
File path functions	**Purpose**	**Supported**
abspath	Gets the absolute path for each item in a list.	VS 2022 17.1
basename	Gets the base name for each item in a list.	VS 2022 17.1

See also

Macros and NMAKE

`abspath` NMAKE function

Article • 02/25/2022

Gets the absolute path for each item in a list.

Syntax

makefile

```
$(abspath input)
```

Parameters

`input`

The list of file paths to convert.

Return value

A list with each of the items from *`input`* converted to their absolute form.

Remarks

`abspath` supports extended-length paths, either by using the `\\?\` prefix, or when long paths are enabled. For more information about long paths, see Maximum Path Length Limitation.

This macro function is available starting in Visual Studio 2022 version 17.1, in NMAKE version 14.31 or later.

Example

makefile

```
$(abspath relative\path\file.c) # If run from "c:\temp", evaluates to
"c:\temp\relative\path\file.c"
$(abspath c:\temp\..\file1.cpp c:\\temp\/dir//) # Evaluates to "c:\file1.cpp
c:\temp\dir\". Follows path traversals and normalizes directory separators.

# abspath can be combined with filter to find which items exist within a
```

```
directory tree
TEMP_SOURCES=$(filteri c:\temp\\%,$(abspath $(SOURCES)))
```

See also

Macros and NMAKE

NMAKE functions by category

`basename` NMAKE function

Article • 02/25/2022

Gets the base name for each item in a list.

Syntax

makefile

```
$(basename input)
```

Parameters

`input`

The list of file paths to convert.

Return value

A list with each of the items from *`input`* converted to their base name (that is, with their extensions removed).

Remarks

`basename` doesn't have any maximum path limitations.

The `basename` function is equivalent to using the R modifier in a filename macro.

This macro function is available starting in Visual Studio 2022 version 17.1, in NMAKE version 14.31 or later.

Example

makefile

```
$(basename c:\temp\file.txt) # Evaluates to "c:\temp\file"
$(basename c:\temp\ c:\file) # Evaluates to "c:\temp\ c:\file" - Directories
and files without extensions are left as-is
$(basename c:\src\.gitignore) # Evaluates to "c:\src\" - Dot files are
considered to be extensions and so are removed
```

See also

Macros and NMAKE

NMAKE functions by category

`filter`, `filteri` NMAKE functions

Article • 11/22/2021

Evaluates to a list of items that matched at least one pattern.

Syntax

```makefile
$(filter filters,input)
$(filteri filters,input)
```

Parameters

`filters`

A list of one or more patterns to filter by.

`input`

The list to be filtered.

Return value

A list of all of the items in *`input`* that match at least one pattern in *`filters`*.

Remarks

`filteri` is the case-insensitive version of `filter`.

This macro function is available starting in Visual Studio 2022, in NMAKE version 14.30 or later.

Example

```makefile
$(filter He%,Hello Hey Hi) # Evaluates to "Hello Hey" - "Hi" doesn't match
the filter
$(filter %y %i,Hello Hey Hi) # Evaluates to "Hey Hi" - items are kept if
they match any filter, "Hello" is dropped as it doesn't match any
$(filter Not%Found,Hello Hey Hi) # Evaluates to "" - none of the items match
```

```
any filters

$(filter he%,Hello Hey Hi) # Evaluates to "" - filter is case-sensitive
$(filteri he%,Hello Hey Hi) # Evaluates to "Hello Hey" - filteri is case-
insensitive

# filteri is commonly used to filter a list of files by their extensions
CPP_SOURCES=$(filteri %.cpp %.cxx,$(SOURCES))
C_SOURCES=$(filteri %.c,$(SOURCES))
```

See also

Macros and NMAKE

NMAKE functions by category

filterout, filterouti

`filterout`, `filterouti` NMAKE functions

Article • 11/22/2021

Evaluates to a list of items that don't match any patterns.

Syntax

makefile

```
$(filterout filters,input)
$(filterouti filters,input)
```

Parameters

`filters`

A list of one or more patterns to filter by.

`input`

The list to be filtered.

Return value

A list of all of the items in *`input`* that don't match any patterns in *`filters`*.

Remarks

`filterouti` is the case-insensitive version of `filterout`.

This macro function is available starting in Visual Studio 2022, in NMAKE version 14.30 or later.

Example

makefile

```
$(filterout He%,Hello Hey Hi) # Evaluates to "Hi" - "Hello" and "Hey" match
the filter
$(filterout %y %i,Hello Hey Hi) # Evaluates to "Hello" - items are kept if
they don't match any filters, "Hey" and "Hi" each match one filter
$(filterout H%,Hello Hey Hi) # Evaluates to "" - each of the items matched
```

```
the filter

$(filterout he%,Hello Hey Hi) # Evaluates to "Hello Hey Hi" - filterout is
case-sensitive
$(filterouti he%,Hello Hey Hi) # Evaluates to "Hi" - filterouti is case-
insensitive
```

See also

Macros and NMAKE

NMAKE functions by category

filter, filteri

`findstring`, `findstringi` NMAKE functions

Article • 11/22/2021

Evaluates to the searched-for string if it's found within another string.

Syntax

makefile

```
$(findstring searchFor,input)
$(findstringi searchFor,input)
```

Parameters

`searchFor`

The string to search for.

`input`

The string to search in.

Return value

If *`searchFor`* is found within *`input`*, then the function returns *`searchFor`*, otherwise it returns null.

Remarks

`findstringi` is the case-insensitive version of `findstring`.

This macro function is available starting in Visual Studio 2022, in NMAKE version 14.30 or later.

Example

makefile

```
$(findstring Hello,Hello World!) # Evaluates to "Hello"
$(findstring Hey,Hello World!) # Evaluates to ""

$(findstring hello,Hello World!) # Evaluates to "" - findstring is case-
sensitive
$(findstringi hello,Hello World!) # Evaluates to "hello" - findstringi is
case-insensitive
```

See also

Macros and NMAKE

NMAKE functions by category

`lowercase` NMAKE function

Article • 02/25/2022

Evaluates to a string where all characters have been converted to their lowercase equivalent.

Syntax

```makefile
$(lowercase input)
```

Parameters

`input`
The string to convert.

Return value

Returns *`input`*, but all characters have been converted to their lowercase equivalent.

Remarks

This macro function is available starting in Visual Studio 2022 version 17.2, in NMAKE version 14.32 or later.

Example

```makefile
$(lowercase Hello World!) # Evaluates to "hello world!"
```

See also

Macros and NMAKE
NMAKE functions by category

`patsubst`, `patsubsti` NMAKE functions

Article • 02/25/2022

Evaluates to a list of items with each item that matches a pattern replaced by a substitution, and items that don't match kept as-is.

Syntax

makefile

```
$(patsubst pattern,replacement,input)
$(patsubsti pattern,replacement,input)
```

Parameters

`pattern`
The pattern to search for.

`replacement`
The pattern to replace *`pattern`* with. If a wildcard is present in *`replacement`*, then it will be replaced with the text that the wildcard in *`pattern`* matched.

`input`
The list of items to be replaced or kept.

Return value

Returns *`input`*, but each item that matches *`pattern`* is replaced by *`replacement`*. Items that don't match *`pattern`* are kept as-is.

Remarks

`patsubsti` is the case-insensitive version of `patsubst`.

This macro function is available starting in Visual Studio 2022 version 17.1, in NMAKE version 14.31 or later.

Example

makefile

```
$(patsubst He%,_%_,Hello Hey Hi) # Evaluates to "_llo_ _y_ Hi"
# "He" matches "Hello" and "Hey", and so "llo" and "y" are matched by the
wildcard
# and used to substitute the wildcard in the replacement. "Hi" is not
matched and so is kept as-is

$(patsubst Hi,Bye,Hello Hey Hi) # Evaluates to "Hello Hey Bye" - No wildcard
is required
$(patsubst %lo,Bye,Hello Hey Hi) # Evaluates to "Bye Hey Hi"
# A wildcard can be used in the pattern without a wildcard in the
replacement

$(patsubst he%,_%_,Hello Hey Hi) # Evaluates to "Hello Hey Hi" - patsubst is
case-sensitive, so no substitutions performed
$(patsubsti he%,_%_,Hello Hey Hi) # Evaluates to "_llo_ _y_ Hi" - patsubsti
is case-insensitive

# patsubsti is commonly used to change the file extensions of a list of
files
OBJ_FILES=$(patsubst %.c,%.obj,$(C_SOURCES)) $(patsubst
%.cpp,%.obj,$(patsubst %.cxx,%.obj,$(CPP_SOURCES)))
```

See also

Macros and NMAKE

NMAKE functions by category

`strip` NMAKE function

Article • 11/22/2021

Cleans up whitespace in and around a list of items.

Syntax

makefile

```
$(strip input)
```

Parameters

`input`

The list to be cleaned.

Return value

A list of the exact same items as *`input`*.

Remarks

NMAKE outputs a list that has a single space between each item and no leading or trailing whitespace. `strip` doesn't change any item within a list, but it does ensure that the returned list is in this canonical form. The canonical form can be useful for later operations that operate on strings instead of lists.

This macro function is available starting in Visual Studio 2022, in NMAKE version 14.30 or later.

Example

makefile

```
$(strip a   b   c d    ) # Evaluates to "a b c d"

# strip is useful to get a canonical form of a list, which can then be
transformed into a different format
SINGLESPACE=$(subst ',,' ') # Use "subst" since a normal assignment trims
```

```
trailing whitespace.
INCLUDE_PATH=$(subst $(SINGLESPACE),;,$(strip $(INCLUDES)))
```

See also

Macros and NMAKE

NMAKE functions by category

`subst`, `substi` NMAKE functions

Article • 11/22/2021

Evaluates to a string where all instances of one string have been replaced with another.

Syntax

makefile

```
$(subst oldString,newString,input)
$(substi oldString,newString,input)
```

Parameters

`oldString`

The string to replace.

`newString`

The string that replaces *`oldString`*. This argument can be null.

`input`

The string to search.

Return value

Returns *`input`*, but all instances of *`oldString`* are replaced by *`newString`*. If *`newString`* is null, then all instances of *`oldString`* are removed.

Remarks

`substi` is the case-insensitive version of `subst`.

This macro function is available starting in Visual Studio 2022, in NMAKE version 14.30 or later.

Example

makefile

```
$(subst Hello,Hey,Hello World!) # Evaluates to "Hey World!"
$(subst ed,ing,red ring mended) # Evaluates to "ring ring mending"
$(subst Hello ,,Hello World!) # Evaluates to "World!"

$(subst hello,Hey,Hello World!) # Evaluates to "Hello World!" - subst is
case-sensitive, so no substitution performed
$(substi hello,Hey,Hello World!) # Evaluates to "Hey World!" - substi is
case-insensitive
```

See also

Macros and NMAKE

NMAKE functions by category

`uppercase` NMAKE function

Article • 02/25/2022

Evaluates to a string where all characters have been converted to their uppercase equivalent.

Syntax

```makefile
$(uppercase input)
```

Parameters

`input`
The string to convert.

Return value

Returns *`input`*, but all characters have been converted to their uppercase equivalent.

Remarks

This macro function is available starting in Visual Studio 2022 version 17.2, in NMAKE version 14.32 or later.

Example

```makefile
$(uppercase Hello World!) # Evaluates to "HELLO WORLD!"
```

See also

Macros and NMAKE
NMAKE functions by category

Special NMAKE macros

Article • 06/21/2022

NMAKE provides several special macros to represent various filenames and commands. One use for some of these macros is in the predefined inference rules. Like all macros, the macros provided by NMAKE are case sensitive.

Filename Macros

Filename macros are predefined as filenames specified in the dependency (not full filename specifications on disk). These macros don't need to be enclosed in parentheses when invoked; specify only a `$` as shown.

Macro	Meaning
`$@`	Current target's full name (path, base name, extension), as currently specified.
`$$@`	Current target's full name (path, base name, extension), as currently specified. Valid only as a dependent in a dependency.
`$*`	Current target's path and base name minus file extension.
`$**`	All dependents of the current target.
`$?`	All dependents with a later timestamp than the current target.
`$<`	Dependent file with a later timestamp than the current target. Valid only in commands in inference rules.

To specify part of a predefined filename macro, append a macro modifier and enclose the modified macro in parentheses.

Modifier	Resulting filename part
`D`	Drive plus directory
`B`	Base name
`F`	Base name plus extension
`R`	Drive plus directory plus base name

Recursion macros

Use recursion macros to call NMAKE recursively. Recursive sessions inherit command-line and environment-variable macros and `Tools.ini` information. They don't inherit makefile-defined inference rules or `.SUFFIXES` and `.PRECIOUS` specifications. There are three ways to pass macros to a recursive NMAKE session:

- Set an environment variable with a `SET` command before the recursive call.
- Define a macro in the command for the recursive call.
- Or, define a macro in `Tools.ini`.

Macro	Definition
`MAKE`	Command used originally to invoke NMAKE. The `$(MAKE)` macro gives the full path to `nmake.exe`.
`MAKEDIR`	Current directory when NMAKE was invoked.
`MAKEFLAGS`	Options currently in effect. Use as `/$(MAKEFLAGS)`. The `/F` option isn't included.

Command macros and options macros

Command macros are predefined for Microsoft products. Options macros represent options to these products and are undefined by default. Both are used in predefined inference rules and can be used in description blocks or user-defined inference rules. Command macros can be redefined to represent part or all of a command line, including options. Options macros generate a null string if left undefined.

Tool	Command macro	Defined as	Options macro
Macro Assembler	`AS`	`ml` or `ml64`	`AFLAGS`
C Compiler	`CC`	`cl`	`CFLAGS`
C++ Compiler	`CPP`	`cl`	`CPPFLAGS`
C++ Compiler	`CXX`	`cl`	`CXXFLAGS`
Resource Compiler	`RC`	`rc`	`RFLAGS`

Environment-variable macros

NMAKE inherits macro definitions for environment variables that exist before the start of the session. If a variable was set in the operating-system environment, it is available as an NMAKE macro. The inherited names are converted to uppercase. Inheritance occurs

before preprocessing. Use the /E option to cause macros inherited from environment variables to override any macros with the same name in the makefile.

Environment-variable macros can be redefined in the session, and this changes the corresponding environment variable. You can also change environment variables with the SET command. Using the SET command to change an environment variable in a session does not change the corresponding macro, however.

For example:

makefile

```
PATH=$(PATH);\nonesuch

all:
    echo %%PATH%%
```

In this example, changing `PATH` changes the corresponding environment variable `PATH`; it appends `\nonesuch` to your path.

If an environment variable is defined as a string that would be syntactically incorrect in a makefile, no macro is created and no warning is generated. If a variable's value contains a dollar sign ($), NMAKE interprets it as the beginning of a macro invocation. Using the macro can cause unexpected behavior.

See also

Macros and NMAKE

Inference rules

Article • 09/30/2021

Inference rules in NMAKE supply commands to update targets and to infer dependents for targets. Extensions in an inference rule match a single target and dependent that have the same base name. Inference rules are user-defined or predefined; predefined rules can be redefined.

If an out-of-date dependency has no commands, and if .SUFFIXES contains the dependent's extension, NMAKE uses a rule whose extensions match the target and an existing file in the current or specified directory. If more than one rule matches existing files, the `.SUFFIXES` list determines which to use; list priority descends from left to right. If a dependent file doesn't exist and isn't listed as a target in another description block, an inference rule can create the missing dependent from another file that has the same base name. If a description block's target has no dependents or commands, an inference rule can update the target. Inference rules can build a command-line target even if no description block exists. NMAKE may invoke a rule for an inferred dependent even if an explicit dependent is specified.

Defining a rule

The *from_ext* represents the extension of a dependent file, and *to_ext* represents the extension of a target file.

makefile

```
.from_ext.to_ext:
   commands
```

Extensions aren't case-sensitive. Macros can be invoked to represent *from_ext* and *to_ext*; the macros are expanded during preprocessing. The period (`.`) that precedes *from_ext* must appear at the beginning of the line. The colon (`:`) is preceded by zero or more spaces or tabs. It can be followed only by spaces or tabs, a semicolon (`;`) to specify a command, a number sign (`#`) to specify a comment, or a newline character. No other spaces are allowed. Commands are specified as in description blocks.

Search paths in rules

makefile

```
{from_path}.from_ext{to_path}.to_ext:
   commands
```

An inference rule applies to a dependency only if paths specified in the dependency exactly match the inference-rule paths. Specify the dependent's directory in *from_path* and the target's directory in *to_path*; no spaces are allowed. Specify only one path for each extension. A path on one extension requires a path on the other. To specify the current directory, use either a period (`.`) or empty braces (`{ }`). Macros can represent *from_path* and *to_path*; they are invoked during preprocessing.

Example of search paths

makefile

```
{dbi\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUDBI) $<

{ilstore\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $<

{misc\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUPDB) $<

{misc\}.c{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $<

{msf\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $<

{bsc\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUPDB) $<

{mre\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUPDB) $<

{namesrvr\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $(YUPDB) $<

{src\cvr\}.cpp{$(ODIR)}.obj::
        $(CC) $(CFLAGS) $<
```

Batch-mode rules

makefile

```
{from_path}.from_ext{to_path}.to_ext::
   commands
```

Batch-mode inference rules provide only one invocation of the inference rule when N commands go through this inference rule. Without batch-mode inference rules, it would require N commands to be invoked. N is the number of dependents that trigger the inference rule.

The only syntactical difference from the standard inference rule is that a batch-mode inference rule ends with a double colon (`::`).

> ⓘ **Note**
>
> The tool being invoked must be able to handle multiple files. The batch-mode inference rule must use `$<` as the macro to access dependent files.

The batch-mode inference rules can speed up the build process. It's faster to supply files to the compiler in batch mode, because the compiler driver is invoked only once. For example, the C and C++ compiler runs faster when handling a set of files, because it can remain memory resident during the entire process.

The following example shows how to use batch-mode inference rules:

makefile

```
#
# sample makefile to illustrate batch-mode inference rules
#
O = .
S = .
Objs = $O/foo1.obj $O/foo2.obj $O/foo2.obj $O/foo3.obj $O/foo4.obj
CFLAGS = -nologo

all : $(Objs)

!ifdef NOBatch
{$S}.cpp{$O}.obj:
!else
{$S}.cpp{$O}.obj::
!endif
   $(CC) $(CFLAGS) -Fd$O\ -c $<

$(Objs) :

#end of makefile
```

NMAKE produces the following output without batch-mode inference rules:

Windows Command Prompt

```
E:\tmp> nmake -f test.mak -a NOBatch=1

Microsoft (R) Program Maintenance Utility   Version 7.00.0000
Copyright (C) Microsoft Corp 1988-2001. All rights reserved.
        cl -nologo -Fd.\ -c .\foo1.cpp
foo1.cpp
        cl -nologo -Fd.\ -c .\foo2.cpp
foo2.cpp
        cl -nologo -Fd.\ -c .\foo3.cpp
foo3.cpp
        cl -nologo -Fd.\ -c .\foo4.cpp
foo4.cpp
```

NMAKE produces the following result with the batch-mode inference rules:

Windows Command Prompt

```
E:\tmp> nmake -f test.mak -a

Microsoft (R) Program Maintenance Utility   Version 7.00.0000
Copyright (C) Microsoft Corp 1988-2001. All rights reserved.

        cl -nologo -Fd.\ -c .\foo1.cpp .\foo2.cpp .\foo3.cpp .\foo4.cpp
foo1.cpp
foo2.cpp
foo3.cpp
foo4.cpp
Generating Code...
```

Predefined rules

Predefined inference rules use NMAKE-supplied command and options macros.

Rule	Command	Default action	Batch rule	Platform
`.asm.exe`	`$(AS) $(AFLAGS) $<`	`ml $<`	no	x86
`.asm.obj`	`$(AS) $(AFLAGS) /c $<`	`ml /c $<`	yes	x86
`.asm.exe`	`$(AS) $(AFLAGS) $<`	`ml64 $<`	no	x64
`.asm.obj`	`$(AS) $(AFLAGS) /c $<`	`ml64 /c $<`	yes	x64
`.c.exe`	`$(CC) $(CFLAGS) $<`	`cl $<`	no	all

Rule	Command	Default action	Batch rule	Platform
`.c.obj`	`$(CC) $(CFLAGS) /c $<`	`cl /c $<`	yes	all
`.cc.exe`	`$(CC) $(CFLAGS) $<`	`cl $<`	no	all
`.cc.obj`	`$(CC) $(CFLAGS) /c $<`	`cl /c $<`	yes	all
`.cpp.exe`	`$(CPP) $(CPPFLAGS) $<`	`cl $<`	no	all
`.cpp.obj`	`$(CPP) $(CPPFLAGS) /c $<`	`cl /c $<`	yes	all
`.cxx.exe`	`$(CXX) $(CXXFLAGS) $<`	`cl $<`	no	all
`.cxx.obj`	`$(CXX) $(CXXFLAGS) /c $<`	`cl /c $<`	yes	all
`.rc.res`	`$(RC) $(RFLAGS) /r $<`	`rc /r $<`	no	all

Inferred dependents and rules

NMAKE assumes an inferred dependent for a target if an applicable inference rule exists. A rule applies if:

- *to_ext* matches the target's extension.
- *from_ext* matches the extension of a file that has the target's base name and that exists in the current or specified directory.
- *from_ext* is in .SUFFIXES; no other *from_ext* in a matching rule has a higher `.SUFFIXES` priority.
- No explicit dependent has a higher `.SUFFIXES` priority.

Inferred dependents can cause unexpected side effects. If the target's description block contains commands, NMAKE executes those commands instead of the commands in the rule.

Precedence in inference rules

If an inference rule is defined more than once, NMAKE uses the highest-precedence definition. The following list shows the order of precedence from highest to lowest:

1. An inference rule defined in a makefile; later definitions have precedence.
2. An inference rule defined in *`Tools.ini`*; later definitions have precedence.

3. A predefined inference rule.

See also

NMAKE reference

Dot directives

Article • 09/30/2021

Specify dot directives outside a description block, at the start of a line. Dot directives begin with a period (`.`) and are followed by a colon (`:`). Spaces and tabs are allowed. Dot directive names are case-sensitive and must be uppercase.

Directive	Purpose
`.IGNORE :`	Ignores nonzero exit codes returned by commands, from the place it is specified to the end of the makefile. By default, NMAKE halts if a command returns a nonzero exit code. To restore error checking, use `!CMDSWITCHES`. To ignore the exit code for a single command, use the dash (`-`) modifier. To ignore exit codes for an entire file, use `/I`.
`.PRECIOUS :` *targets*	Preserves *targets* on disk if the commands to update them are halted; has no effect if a command handles an interrupt by deleting the file. Separate the target names with one or more spaces or tabs. By default, NMAKE deletes a target if a build is interrupted by CTRL+C or CTRL+BREAK. Each use of `.PRECIOUS` applies to the entire makefile; multiple specifications are cumulative.
`.SILENT`	Suppresses display of executed commands, from the place it is specified to the end of the makefile. By default, NMAKE displays the commands it invokes. To restore echoing, use `!CMDSWITCHES`. To suppress echoing of a single command, use the `@` modifier. To suppress echoing for an entire file, use `/S`.
`.SUFFIXES :` *list*	Lists extensions for inference-rule matching; predefined to include the following extensions: `.exe .obj .asm .c .cpp .cxx .bas .cbl .for .pas .res .rc .f .f90`

To change the `.SUFFIXES` list order or to specify a new list, clear the list and specify a new setting. To clear the list, specify no extensions after the colon:

makefile

```
.SUFFIXES :
```

To add additional suffixes to the end of the list, specify

makefile

```
.SUFFIXES : suffix_list
```

where *suffix_list* is a list of the additional suffixes, separated by one or more spaces or tabs. To see the current setting of `.SUFFIXES`, run NMAKE with `/P`.

See also

NMAKE Reference

Makefile preprocessing

Article • 02/17/2022

You can control the NMAKE session by using preprocessing directives and expressions. Preprocessing instructions can be placed in the makefile or in `Tools.ini`. Using directives, you can conditionally process your makefile, display error messages, include other makefiles, undefine a macro, and turn certain options on or off.

Makefile Preprocessing Directives

Preprocessing directives aren't case-sensitive. The initial exclamation point (`!`) must appear at the beginning of the line. Zero or more spaces or tabs can appear after the exclamation point, for indentation.

- **`!CMDSWITCHES`** { **`+`***option* | **`-`***option* } ...

 Turns each listed *option* on or off. Spaces or tabs must appear before the `+` or `-` operator. No spaces can appear between the operator and the option letters. Letters aren't case-sensitive and are specified without a slash (`/`). To turn on some options and turn off others, use separate specifications of **`!CMDSWITCHES`**.

 Only **`/D`**, **`/I`**, **`/N`**, and **`/S`** can be used in a makefile. In `Tools.ini`, all options are allowed except **`/F`**, **`/HELP`**, **`/NOLOGO`**, **`/X`**, and **`/?`**. Changes specified in a description block don't take effect until the next description block. This directive updates **`MAKEFLAGS`**; changes are inherited during recursion if **`MAKEFLAGS`** is specified.

- **`!ERROR`** *text*

 Displays *text* in error U1050, then halts NMAKE, even if **`/K`**, **`/I`**, **`.IGNORE`**, **`!CMDSWITCHES`**, or the dash (`-`) command modifier is used. Spaces or tabs before *text* are ignored.

- **`!MESSAGE`** *text*

 Displays *text* to standard output. Spaces or tabs before *text* are ignored.

- **`!INCLUDE`** [**`<`**] *filename* [**`>`**]

 Reads *filename* as a makefile, then continues with the current makefile. NMAKE searches for *filename* first in the specified or current directory, then recursively through directories of any parent makefiles, then, if *filename* is enclosed by angle

brackets (`< >`), in directories specified by the **`INCLUDE`** macro, which is initially set to the `INCLUDE` environment variable. Useful to pass `.SUFFIXES` settings, `.PRECIOUS`, and inference rules to recursive makefiles.

- **`!IF`** *constant_expression*

 Processes statements between **`!IF`** and the next **`!ELSE`** or **`!ENDIF`** if *constant_expression* evaluates to a nonzero value.

- **`!IFDEF`** *macro_name*

 Processes statements between **`!IFDEF`** and the next **`!ELSE`** or **`!ENDIF`** if *macro_name* is defined. A null macro is considered to be defined.

- **`!IFNDEF`** *macro_name*

 Processes statements between **`!IFNDEF`** and the next **`!ELSE`** or **`!ENDIF`** if *macro_name* isn't defined.

- **`!ELSE`** [**`IF`** *constant_expression* | **`IFDEF`** *macro_name* | **`IFNDEF`** *macro_name*]

 Processes statements between **`!ELSE`** and the next **`!ENDIF`** if the prior **`!IF`**, **`!IFDEF`**, or **`!IFNDEF`** statement evaluated to zero. The optional keywords give further control of preprocessing.

- **`!ELSEIF`**

 Synonym for **`!ELSE IF`**.

- **`!ELSEIFDEF`**

 Synonym for **`!ELSE IFDEF`**.

- **`!ELSEIFNDEF`**

 Synonym for **`!ELSE IFNDEF`**.

- **`!ENDIF`**

 Marks the end of an **`!IF`**, **`!IFDEF`**, or **`!IFNDEF`** block. Any text after **`!ENDIF`** on the same line is ignored.

- **`!UNDEF`** *macro_name*

 Undefines *macro_name*.

Expressions in makefile preprocessing

The `!IF` or `!ELSE IF` *constant_expression* consists of integer constants (in decimal or C-language notation), string constants, or commands. Use parentheses to group expressions. Expressions use C-style signed long integer arithmetic; numbers are in 32-bit two's-complement form in the range -2147483648 to 2147483647.

Expressions can use operators that act on constant values, exit codes from commands, strings, macros, and file-system paths.

Makefile preprocessing operators

Makefile preprocessing expressions can use operators that act on constant values, exit codes from commands, strings, macros, and file-system paths. To evaluate the expression, the preprocessor first expands macros, and then executes commands, and then does the operations. It evaluates operations in order of explicit grouping in parentheses, and then in order of operator precedence. The result is a constant value.

The `DEFINED` operator is a logical operator that acts on a macro name. The expression `DEFINED( macro_name )` is true if *macro_name* is defined, even if it doesn't have an assigned value. `DEFINED` in combination with `!IF` or `!ELSE IF` is equivalent to `!IFDEF` or `!ELSE IFDEF`. However, unlike these directives, `DEFINED` can be used in complex expressions.

The `EXIST` operator is a logical operator that acts on a file-system path. `EXIST( path )` is true if *path* exists. The result from `EXIST` can be used in binary expressions. If *path* contains spaces, enclose it in double quotation marks.

To compare two strings, use the equality (`==`) operator or the inequality (`!=`) operator. Enclose strings in double quotation marks.

Integer constants can use the unary operators for numerical negation (`-`), one's complement (`~`), and logical negation (`!`).

Expressions can use the following operators. The operators of equal precedence are grouped together, and the groups are listed in decreasing order of precedence. Unary operators associate with the operand to the right. Binary operators of equal precedence associate operands from left to right.

Operator	Description

Operator	Description
`DEFINED(` *macro_name* `)`	Produces a logical value for the current definition state of *macro_name*.
`EXIST(` *path* `)`	Produces a logical value for the existence of a file at *path*.
`!`	Unary logical NOT.
`~`	Unary one's complement.
`-`	Unary negation.
`*`	Multiplication.
`/`	Division.
`%`	Modulus (remainder).
`+`	Addition.
`-`	Subtraction.
`<<`	Bitwise shift left.
`>>`	Bitwise shift right.
`<=`	Less than or equal.
`>=`	Greater than or equal.
`<`	Less than.
`>`	Greater than.
`==`	Equality.
`!=`	Inequality.
`&`	Bitwise AND.
`^`	Bitwise XOR.
`\|`	Bitwise OR.

Operator	Description
`&&`	Logical AND.
`\|\|`	Logical OR.

> **Note**
>
> The bitwise XOR operator (`^`) is the same as the escape character, and must be escaped (as `^^`) when it's used in an expression.

Executing a program in preprocessing

To use a command's exit code during preprocessing, specify the command, with any arguments, within brackets (`[ ]`). Any macros are expanded before the command is executed. NMAKE replaces the command specification with the command's exit code, which can be used in an expression to control preprocessing.

Example

Makefile

```
!IF [my_command.exe arg1 arg2] != 0
!MESSAGE my_command.exe failed!
!ENDIF
```

See also

NMAKE Reference

LIB Reference

Article • 08/03/2021

The Microsoft Library Manager (LIB.exe) creates and manages a library of Common Object File Format (COFF) object files. LIB can also be used to create export files and import libraries to reference exported definitions.

> **Note**
>
> You can start this tool only from the Visual Studio command prompt. You cannot start it from a system command prompt or from File Explorer.

- Overview of LIB
- How to: Set LIB.EXE Options in the Visual Studio Development Environment
- Running LIB
- Managing a Library
- Extracting a Library Member
- Working with Import Libraries and Export Files

See also

Additional MSVC Build Tools

Overview of LIB

Article • 08/03/2021

LIB (lib.exe) creates standard libraries, import libraries, and export files you can use with LINK when building a program. LIB runs from a command prompt.

You can use LIB in the following modes:

- Building or modifying a COFF library
- Extracting a member object to a file
- Creating an export file and an import library

These modes are mutually exclusive; you can use LIB in only one mode at a time.

LIB options

The following table lists the options for lib.exe, with a link to more information.

Option	Description
/DEF	Create an import library and an export file. For more information, see Building an Import Library and Export File.
/ERRORREPORT	Deprecated. For more information, see Running LIB.
/EXPORT	Exports a function from your program. For more information, see Building an Import Library and Export File.
/EXTRACT	Create an object (.obj) file that contains a copy of a member of an existing library. For more information, see Extracting a Library Member.
/INCLUDE	Adds a symbol to the symbol table. For more information, see Building an Import Library and Export File.
/LIBPATH	Overrides the environment library path. For more information, see Managing a Library.

Option	Description
/LINKREPRO	Creates artifacts needed to reproduce a lib.exe crash or internal error. For more information, see Running LIB.
/LINKREPROTARGET	Only generates the **/LINKREPRO** artifacts when lib.exe is used with a specified file. For more information, see Running LIB.
/LIST	Displays information about the output library to standard output. For more information, see Managing a Library.
/LTCG	Causes the library to be built using link-time code generation. For more information, see Running LIB.
/MACHINE	Specifies the target platform for the program. For more information, see Running LIB.
/NAME	When building an import library, specifies the name of the DLL for which the import library is being built. For more information, see Managing a Library.
/NODEFAULTLIB	Removes one or more default libraries from the list of libraries it searches when resolving external references. For more information, see Managing a Library.
/NOLOGO	Suppresses display of the LIB copyright message and version number and prevents echoing of command files. For more information, see Running LIB.
/OUT	Overrides the default output filename. For more information, see Managing a Library.
/REMOVE	Omits an object from the output library. For more information, see Managing a Library.
/SUBSYSTEM	Tells the operating system how to run a program created by linking to the output library. For more information, see Managing a Library.

Option	Description
/VERBOSE	Displays details about the progress of the session, including names of the .obj files being added. For more information, see Running LIB.
/WX	Treat warnings as errors. For more information, see Running LIB.

See also

LIB Reference
LIB Input Files
LIB Output Files
Other LIB Output
Structure of a Library

How to: Set LIB.EXE Options in the Visual Studio Development Environment

Article • 08/03/2021

To set LIB.EXE options in the Visual Studio development environment

1. Access the project's Property Page dialog box.
2. With a static library project active, select the **Librarian** node.
3. Select either the General or Input/Output property page.
4. Modify properties as needed.

See also

LIB Reference

LIB Input Files

Article • 08/03/2021

The input files expected by LIB depend on the mode in which it is being used, as shown in the following table.

Mode	Input
Default (building or modifying a library)	COFF object (.obj) files, COFF libraries (.lib), 32-bit Object Model Format (OMF) object (.obj) files
Extracting a member with /EXTRACT	COFF library (.lib)
Building an export file and import library with /DEF	Module-definition (.def) file, COFF object (.obj) files, COFF libraries (.lib), 32-bit OMF object (.obj) files

> **Note**
>
> OMF libraries created by the 16-bit version of LIB cannot be used as input to the 32-bit version of LIB.

See also

Overview of LIB

LIB Output Files

Article • 08/03/2021

The output files produced by LIB depend on the mode in which it is being used, as shown in the following table.

Mode	Output
Default (building or modifying a library)	COFF library (.lib)
Extracting a member with /EXTRACT	Object (.obj) file
Building an export file and import library with /DEF	Import library (.lib) and export (.exp) file

See also

Overview of LIB

Other LIB Output

Article • 08/03/2021

In the default mode, you can use the /LIST option to display information about the resulting library. You can redirect this output to a file.

LIB displays a copyright and version message and echoes command files unless the /NOLOGO option is used.

When you type `lib` with no other input, LIB displays a usage statement that summarizes its options.

Error and warning messages issued by LIB have the form LNK*nnnn*. The LINK, DUMPBIN, and EDITBIN tools also use this range of errors. Help is available by selecting the error in the Output window and pressing F1.

See also

Overview of LIB

Structure of a Library

Article • 08/03/2021

A library contains COFF objects. Objects in a library contain functions and data that can be referenced externally by other objects in a program. An object in a library is sometimes referred to as a library member.

You can get additional information about the contents of a library by running the DUMPBIN tool with the /LINKERMEMBER option. For more information about this option, see DUMPBIN Reference.

See also

Overview of LIB

Running LIB

Article • 02/17/2022

Various command-line options can be used to control LIB.

LIB Command Line

To run LIB, type the command `lib`, followed by the options and file names for the task you're using LIB for. LIB also accepts command-line input in command files, which are described in the following section. LIB doesn't use an environment variable.

LIB Command Files

You can pass command-line arguments to LIB in a command file using the following syntax:

> **LIB** @*command-file*

The file *command-file* is a text file. No spaces or tabs are allowed between the at sign (@) and the file name. The *command-file* name has no default extension. Specify the full file name, including any extension. Wildcards can't be used. You may specify an absolute or relative path with the file name.

In the command file, arguments can be separated by spaces or tabs, as they can on the command line. Arguments can also be separated by newline characters. Use a semicolon (;) to mark a comment. LIB ignores all text from the semicolon to the end of the line.

You can specify either all or part of the command line in a command file, and you may use more than one command file in a LIB command. LIB accepts the command-file input as if it's specified in that location on the command line. Command files can't be nested. LIB echoes the contents of command files unless the **/NOLOGO** option is used.

Using LIB Options

An option consists of an option specifier, which is either a dash (-) or a forward slash (/), followed by the name of the option. Option names can't be abbreviated. Some options take an argument, specified after a colon (:). No spaces or tabs are allowed within an option specification. Use one or more spaces or tabs to separate option specifications on the command line. Option names and their keyword or file name arguments aren't

case-sensitive, but identifiers used as arguments are case-sensitive. LIB processes options in the order specified on the command line and in command files. If an option is repeated with different arguments, the last one to be processed takes precedence.

The following options apply to all modes of LIB:

> **/ERRORREPORT** [**NONE** | **PROMPT** | **QUEUE** | **SEND**]

The /ERRORREPORT option is deprecated. Starting in Windows Vista, error reporting is controlled by Windows Error Reporting (WER) settings.

> **/LINKREPRO:***directory-path*
> **/LINKREPROTARGET:***filename*

To help Microsoft diagnose lib.exe crashes and internal errors, you can use the /LINKREPRO option. This option generates a *link repro*, a set of build artifacts that allow Microsoft to reproduce a problem that occurs during library operations. The /LINKREPROTARGET option can be used with the **/LINKREPRO** option. It only generates link repro artifacts when lib.exe produces the specified file. For more information, see How to report a problem with the Microsoft C++ toolset.

> **/LTCG**

"LTCG" stands for *link-time code generation*. This feature requires cooperation between the compiler (cl.exe), LIB, and the linker (LINK). Together they can optimize code beyond what any component can do by itself.

The **/LTCG** option to LIB specifies that the inputs from cl.exe include object files generated by using the /GL compiler option. If LIB encounters such inputs, and **/LTCG** isn't specified, it restarts with /LTCG enabled after displaying an informational message. In other words, it isn't necessary to set this option explicitly, but it speeds up build performance. That's because LIB doesn't have to restart itself.

In the build process, the output from LIB is sent to LINK. LINK has its own separate **/LTCG** option. It's used to perform various optimizations, including whole-program optimization and profile-guided optimization (PGO) instrumentation. For more information about the LINK option, see /LTCG.

> **/MACHINE**

Specifies the target platform for the program. Usually, you don't need to specify **/MACHINE**. LIB infers the machine type from the .obj files. However, in some

circumstances, LIB can't determine the machine type and issues an error message. If such an error occurs, specify **/MACHINE**. In **/EXTRACT** mode, this option is for verification only. Use `lib /?` at the command line to see available machine types.

> **/NOLOGO**

Suppresses display of the LIB copyright message and version number and prevents echoing of command files.

> **/VERBOSE**

Displays details about the progress of the session, including names of the .obj files being added. The information is sent to standard output and can be redirected to a file.

> **/WX**[**:NO**]

Treat warnings as errors. For more information, see /WX (Treat Linker Warnings as Errors).

Other options apply only to specific modes of LIB. These options are discussed in the sections describing each mode.

See also

LIB Reference

Managing a Library

Article • 03/03/2022

The default mode for LIB is to build or modify a library of COFF objects. LIB runs in this mode when you don't specify `/EXTRACT` (to copy an object to a file) or `/DEF` (to build an import library).

To build a library from objects and/or libraries, use the following syntax:

> **LIB** [*options...*] *files...*

This command creates a library from one or more input files, *files*. The *files* can be COFF object files, 32-bit OMF object files, or existing COFF libraries. LIB creates one library that contains all objects in the specified files. If an input file is a 32-bit OMF object file, LIB converts it to COFF before building the library. LIB can't accept a 32-bit OMF object that's in a library created by the 16-bit version of LIB. You must first use the 16-bit LIB to extract the object; then you can use the extracted object file as input to the 32-bit LIB.

By default, LIB names the output file using the base name of the first object or library file and the extension *.lib*. The output file is put in the current directory. If a file already exists with the same name, the output file replaces the existing file. To preserve an existing library, use the `/OUT` option to specify a name for the output file.

The following options apply to building and modifying a library:

`/LIBPATH:` *dir*
Overrides the environment library path and sets it to *dir*. For details, see the description of the LINK /LIBPATH option.

`/LIST`
Displays information about the output library to standard output. The output can be redirected to a file. You can use `/LIST` to determine the contents of an existing library without modifying it.

`/NAME:` *filename*
When building an import library, *filename* specifies the name of the DLL for which the import library is being built.

`/NODEFAULTLIB`
Removes one or more default libraries from the list of libraries it searches when

resolving external references. For more information, see /NODEFAULTLIB.

`/OUT:` *`filename`*
Overrides the default output filename and replaces it with *`filename`*. By default, the output library is created in the current directory, with the base name of the first library or object file on the command line and the extension *`.lib`*.

`/REMOVE:` *`object`*
Omits the specified *`object`* from the output library. LIB creates an output library by combining all objects (whether in object files or libraries), and then deleting any objects specified with `/REMOVE`.

`/SUBSYSTEM:`{`CONSOLE` | `EFI_APPLICATION` | `EFI_BOOT_SERVICE_DRIVER` | `EFI_ROM` | `EFI_RUNTIME_DRIVER` | `NATIVE` | `POSIX` | `WINDOWS` | `WINDOWSCE`}[,#[.##]]
Tells the operating system how to run a program created by linking to the output library. For more information, see the description of the LINK /SUBSYSTEM option.

LIB options specified on the command line aren't case sensitive.

You can use LIB to perform the following library-management tasks:

- To add objects to a library, specify the file name for the existing library and the filenames for the new objects.
- To combine libraries, specify the library file names. You can add objects and combine libraries with a single LIB command.
- To replace a library member with a new object, specify the library containing the member object to be replaced and the file name for the new object (or the library that contains it). When an object that has the same name exists in more than one input file, LIB puts the last object specified in the LIB command into the output library. When you replace a library member, be sure to specify the new object or library after the library that contains the old object.
- To delete a member from a library, use the /REMOVE option. LIB processes any specifications of /REMOVE after combining all input objects, regardless of command-line order.

> ⓘ **Note**
>
> You can't both delete a member and extract it to a file in the same step. You must first extract the member object using `/EXTRACT`, then run LIB again using `/REMOVE`.

This behavior differs from that of the 16-bit LIB (for OMF libraries) provided in other Microsoft products.

See also

LIB Reference

Extracting a Library Member

Article • 08/03/2021

You can use LIB to create an object (.obj) file that contains a copy of a member of an existing library. To extract a copy of a member, use the following syntax:

```
LIB library /EXTRACT:member /OUT:objectfile
```

This command creates an .obj file called *objectfile* that contains a copy of a `member` of a *library*. The `member` name is case sensitive. You can extract only one member in a single command. The /OUT option is required; there is no default output name. If a file called *objectfile* already exists in the specified directory (or the current directory, if no directory is specified with *objectfile*), the extracted *objectfile* replaces the existing file.

See also

LIB Reference

Working with Import Libraries and Export Files

Article • 08/03/2021

You can use LIB with the /DEF option to create an import library and an export file. LINK uses the export file to build a program that contains exports (usually a dynamic-link library (DLL)), and it uses the import library to resolve references to those exports in other programs.

Note that if you create your import library in a preliminary step, before creating your .dll, you must pass the same set of object files when building the .dll, as you passed when building the import library.

In most situations, you do not need to use LIB to create your import library. When you link a program (either an executable file or a DLL) that contains exports, LINK automatically creates an import library that describes the exports. Later, when you link a program that references those exports, you specify the import library.

However, when a DLL exports to a program that it also imports from, whether directly or indirectly, you must use LIB to create one of the import libraries. When LIB creates an import library, it also creates an export file. You must use the export file when linking one of the DLLs.

See also

LIB Reference

Building an Import Library and Export File

Article • 08/03/2021

To build an import library and export file, use the following syntax:

> **LIB /DEF**[:*deffile*] [*options*] [*objfiles*] [*libraries*]

When /DEF is specified, LIB creates the output files from export specifications that are passed in the LIB command. There are three methods for specifying exports, listed in recommended order of use:

1. A `__declspec(dllexport)` definition in one of the *objfiles* or *libraries*
2. A specification of /EXPORT:*name* on the LIB command line
3. A definition in an **EXPORTS** statement in a *deffile*

These are the same methods you use to specify exports when linking an exporting program. A program can use more than one method. You can specify parts of the LIB command (such as multiple *objfiles* or /EXPORT specifications) in a command file in the LIB command, just as you can in a LINK command.

The following options apply to building an import library and export file:

> **/OUT:** *import*

Overrides the default output file name for the *import* library being created. When /OUT is not specified, the default name is the base name of the first object file or library in the LIB command and the extension .lib. The export file is given the same base name as the import library and the extension .exp.

> **/EXPORT:** *entryname*[**=** *internalname*][,@*ordinal*[, **NONAME**]][, **DATA**]

Exports a function from your program to allow other programs to call the function. You can also export data (using the **DATA** keyword). Exports are usually defined in a DLL.

The *entryname* is the name of the function or data item as it is to be used by the calling program. Optionally, you can specify the *internalname* as the function known in the defining program; by default, *internalname* is the same as *entryname*. The *ordinal* specifies an index into the export table in the range 1 through 65,535; if you do not

specify *ordinal*, LIB assigns one. The **NONAME** keyword exports the function only as an ordinal, without an *entryname*. The **DATA** keyword is used to export data-only objects.

> **/INCLUDE:** *symbol*

Adds the specified *symbol* to the symbol table. This option is useful for forcing the use of a library object that otherwise would not be included.

Note that if you create your import library in a preliminary step, before creating your .dll, you must pass the same set of object files when building the .dll, as you passed when building the import library.

See also

Working with Import Libraries and Export Files

Using an Import Library and Export File

Article • 08/03/2021

When a program (either an executable file or a DLL) exports to another program that it also imports from, or if more than two programs both export to and import from each other, the commands to link these programs must accommodate circular exports.

In a situation without circular exports, when linking a program that uses exports from another program, you must specify the import library for the exporting program. The import library for the exporting program is created when you link that exporting program. Therefore, you must link the exporting program before the importing program. For example, if TWO.dll imports from ONE.dll, you must first link ONE.dll and get the import library ONE.lib. Then, you specify ONE.lib when linking TWO.dll. When the linker creates TWO.dll, it also creates its import library, TWO.lib. Use TWO.lib when linking programs that import from TWO.dll.

However, in a circular export situation, it is not possible to link all of the interdependent programs using import libraries from the other programs. In the example discussed earlier, if TWO.dll also exports to ONE.dll, the import library for TWO.dll won't exist yet when ONE.dll is linked. When circular exports exist, you must use LIB to create an import library and export file for one of the programs.

To begin, choose one of the programs on which to run LIB. In the LIB command, list all objects and libraries for the program and specify /DEF. If the program uses a .def file or /EXPORT specifications, specify these as well.

After you create the import library (.lib) and the export file (.exp) for the program, you use the import library when linking the other program or programs. LINK creates an import library for each exporting program it builds. For example, if you run LIB on the objects and exports for ONE.dll, you create ONE.lib and ONE.exp. You can now use ONE.lib when linking TWO.dll; this step also creates the import library TWO.lib.

Finally, link the program you began with. In the LINK command, specify the objects and libraries for the program, the .exp file that LIB created for the program, and the import library or libraries for the exports used by the program. To continue the example, the LINK command for ONE.dll contains ONE.exp and TWO.lib, as well as the objects and libraries that go into ONE.dll. Do not specify the .def file or /EXPORT specifications in the LINK command; these are not needed, because the export definitions are contained in the .exp file. When you link using an .exp file, LINK does not create an import library, because it assumes that one was created when the .exp file was created.

See also

Working with Import Libraries and Export Files

EDITBIN Reference

Article • 08/03/2021

The Microsoft COFF Binary File Editor (EDITBIN.EXE) modifies Common Object File Format (COFF) binary files. You can use EDITBIN to modify object files, executable files, and dynamic-link libraries (DLL).

> **Note**
>
> You can start this tool only from the Visual Studio command prompt. You cannot start it from a system command prompt or from File Explorer.

EDITBIN is not available for use on files produced with the /GL compiler option. Any modifications to binary files produced with /GL will have to be achieved by recompiling and linking.

- EDITBIN command line
- EDITBIN options

See also

Additional MSVC Build Tools

EDITBIN Command Line

Article • 08/03/2021

To run EDITBIN, use the following syntax:

```
EDITBIN [options] files...
```

Specify one or more files for the objects or images to be changed, and one or more options for changing the files.

When you type the command `editbin` without any other command-line input, EDITBIN displays a usage statement that summarizes its options.

See also

Additional MSVC Build Tools
EDITBIN Reference

EDITBIN Options

Article • 08/03/2021

You can use EDITBIN to modify object files, executable files, and dynamic-link libraries (DLLs). Options specify the changes that EDITBIN makes.

An option consists of an option specifier, which is either a dash (`-`) or a forward slash (`/`), followed by the name of the option. Option names can't be abbreviated. Some options take arguments that are specified after a colon (`:`). No spaces or tabs are allowed within an option specification. Use one or more spaces or tabs to separate option specifications on the command line. Option names and their keyword arguments or file name arguments aren't case-sensitive. For example, `-bind` and `/BIND` mean the same thing.

EDITBIN has the following options:

Option	Purpose
/ALLOWBIND	Specifies whether a DLL can be bound.
/ALLOWISOLATION	Specifies DLL or executable file manifest lookup behavior.
/APPCONTAINER	Specifies whether the app must run within an AppContainer—for example, a UWP app.
/BIND	Sets the addresses for the entry points in the specified objects to speed load time.
/DYNAMICBASE	Specifies whether the DLL or executable image can be randomly rebased at load-time by using address space layout randomization (ASLR).
/ERRORREPORT	Deprecated. Error reporting is controlled by Windows Error Reporting (WER) settings.
/HEAP	Sets the size of the executable image's heap in bytes.
/HIGHENTROPYVA	Specifies whether the DLL or executable image supports high entropy (64-bit) address space layout randomization (ASLR).
/INTEGRITYCHECK	Specifies whether to check the digital signature at load time.
/LARGEADDRESSAWARE	Specifies whether the object supports addresses that are larger than two gigabytes.
/NOLOGO	Suppresses the EDITBIN startup banner.

Option	Purpose
/NXCOMPAT	Specifies whether the executable image is compatible with Windows Data Execution Prevention.
/REBASE	Sets the base addresses for the specified objects.
/RELEASE	Sets the checksum in the header.
/SECTION	Overrides the attributes of a section.
/STACK	Sets the size of the executable image's stack in bytes.
/SUBSYSTEM	Specifies the execution environment.
/SWAPRUN	Specifies that the executable image is copied to the swap file, and then run from there.
/TSAWARE	Specifies that the app is designed to run in a multi-user environment.
/VERSION	Sets the version number in the header.

See also

Additional MSVC build tools
EDITBIN Reference

/ALLOWISOLATION

Article • 11/07/2022

Specifies behavior for manifest lookup.

Syntax

```
/ALLOWISOLATION[:NO]
```

Remarks

/ALLOWISOLATION causes the operating system to do manifest lookups and loads.

/ALLOWISOLATION is the default.

/ALLOWISOLATION:NO indicates that executables are loaded as if there were no manifest, and causes EDITBIN Reference to set the `IMAGE_DLLCHARACTERISTICS_NO_ISOLATION` bit in the optional header's `DllCharacteristics` field.

When isolation is disabled for an executable, the Windows loader doesn't try to find an application manifest for the newly created process. The new process doesn't have a default activation context, even if there is a manifest in the executable itself or if there is a manifest that has the name *executable-name*.exe.manifest.

See also

EDITBIN Options
/ALLOWISOLATION (Manifest Lookup)
Manifest Files Reference

/ALLOWBIND

Article • 11/07/2022

Specifies whether a DLL can be bound.

```
/ALLOWBIND[:NO]
```

Remarks

The **/ALLOWBIND** option sets a bit in a DLL's header that indicates to Bind.exe that the image is allowed to be bound. Binding can allow an image to load faster when the loader doesn't have to rebase and perform address fixup for each referenced DLL. You may not want a DLL to be bound if it has been digitally signed—binding invalidates the signature. Binding has no effect if address space layout randomization (ASLR) is enabled for the image by using **/DYNAMICBASE** on versions of Windows that support ASLR.

Use **/ALLOWBIND:NO** to prevent Bind.exe from binding the DLL.

For more information, see the /ALLOWBIND linker option.

See also

EDITBIN Options

/APPCONTAINER

Article • 11/07/2022

Marks an executable that must run in an app container—for example, a Microsoft Store or Universal Windows app.

```
/APPCONTAINER[:NO]
```

Remarks

An executable that has the **/APPCONTAINER** option set can only be run in an app container, which is the process-isolation environment introduced in Windows 8. For Microsoft Store and Universal Windows apps, this option must be set.

See also

EDITBIN Options
What's a Universal Windows App?

/BIND

Article • 08/03/2021

```
/BIND[:PATH=path]
```

Remarks

This option sets the addresses of the entry points in the import address table for an executable file or DLL. Use this option to reduce load time of a program.

Specify the program's executable file and DLLs in the *files* argument on the EDITBIN command line. The optional *path* argument to /BIND specifies the location of the DLLs used by the specified files. Separate multiple directories with a semicolon (;). If *path* is not specified, EDITBIN searches the directories specified in the PATH environment variable. If *path* is specified, EDITBIN ignores the PATH variable.

By default, the program loader sets the addresses of entry points when it loads a program. The amount of time this process takes varies, depending on the number of DLLs and the number of entry points referenced in the program. If a program has been modified with /BIND, and if the base addresses for the executable file and its DLLs do not conflict with DLLs that are already loaded, the operating system does not need to set these addresses. In a situation where the files are incorrectly based, the operating system relocates the program's DLLs and recalculates the entry-point addresses, which adds to the program's load time.

See also

EDITBIN Options

/DYNAMICBASE

Article • 05/06/2022

Specifies whether to generate an executable image that can be randomly rebased at load time by using the address space layout randomization (ASLR) feature of Windows that was first available in Windows Vista.

Syntax

> `/DYNAMICBASE`[`:NO`]

Remarks

The `/DYNAMICBASE` option modifies the header of an *executable image*, a .dll or .exe file, to indicate whether the application should be randomly rebased at load time, and enables virtual address allocation randomization, which affects the virtual memory location of heaps, stacks, and other operating system allocations. The `/DYNAMICBASE` option applies to both 32-bit and 64-bit images. ASLR is supported on Windows Vista and later operating systems. The option is ignored by earlier operating systems.

By default, `/DYNAMICBASE` is enabled. To disable this option, use `/DYNAMICBASE:NO`. The `/DYNAMICBASE` option is required for the /HIGHENTROPYVA option to have an effect.

Because ASLR can't be disabled on ARM, ARM64, or ARM64EC architectures, `/DYNAMICBASE:NO` isn't supported for these targets.

See also

- EDITBIN Options
- Windows ISV Software Security Defenses

/ERRORREPORT (editbin.exe)

Article • 08/03/2021

> **Note**
>
> The /ERRORREPORT option is deprecated. Starting in Windows Vista, error reporting is controlled by **Windows Error Reporting (WER)** settings.

Syntax

> **/ERRORREPORT** [**NONE** | **PROMPT** | **QUEUE** | **SEND**]

Remarks

The **/ERRORREPORT** arguments are overridden by the Windows Error Reporting service settings. EDITBIN automatically sends reports of internal errors to Microsoft, if reporting is enabled by Windows Error Reporting. No report is sent if disabled by Windows Error Reporting.

See also

EDITBIN Options

/HEAP

Article • 08/03/2021

Sets the size of the heap in bytes. This option only applies to executable files.

Syntax

> **/HEAP:***reserve*[**,***commit*]

Remarks

The *reserve* argument specifies the total initial heap allocation in virtual memory. The **/HEAP** linker or EDITBIN option rounds up the specified value to the nearest multiple of 4 bytes. By default, the heap size is 1 MB.

The optional *commit* argument is subject to interpretation by the operating system. On a Windows operating system, it specifies the initial amount of physical memory to allocate. It also specifies how much more memory to allocate when the heap is expanded. Committed virtual memory causes space to be reserved in the paging file. A higher *commit* value allows the system to allocate memory less often when the app needs more heap space but increases the memory requirements and possibly the app startup duration. The *commit* value must be less than or equal to the *reserve* value. The default value is 4 KB.

Specify the *reserve* and *commit* values in decimal, C-language hexadecimal, or octal notation. For example, a value of 1 MB can be specified as 1048576 in decimal, or as 0x100000 in hexadecimal, or as 04000000 in octal. The default values are equivalent to the option **/HEAP:1048576,4096**.

Example

This example link command creates an executable *main.exe* that has heap reserve of 2 MB. The initial heap and later heap expansions come in blocks of 64 KB:

```
link /heap:0x200000,0x10000 main.obj
```

To set this linker option in Visual Studio

1. Open the project **Property Pages** dialog box. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **System** property page.
3. Set the **Heap Reserve Size** and **Heap Commit Size** properties, then choose **OK** or **Apply** to save your changes.

See also

EDITBIN options
MSVC linker options

/HIGHENTROPYVA

Article • 08/03/2021

Specifies whether the executable image supports high-entropy 64-bit address space layout randomization (ASLR).

Syntax

> `/HIGHENTROPYVA`[`:NO`]

Remarks

This option modifies the header of an *executable image* file (for example, a `.dll` or `.exe` file), to indicate support for 64-bit address ASLR. To have an effect, set the option on both the executable and all modules that it depends on. Then operating systems that support 64-bit ASLR can rebase the executable image's segments at load time by using randomized 64-bit virtual addresses. This large address space makes it more difficult for an attacker to guess the location of a particular memory region.

By default, the linker enables `/HIGHENTROPYVA` for 64-bit executable images. This option requires both /DYNAMICBASE and /LARGEADDRESSAWARE, which are also enabled by default for 64-bit images. `/HIGHENTROPYVA` isn't applicable to 32-bit executable images, where the option is ignored. To explicitly disable this option, use `/HIGHENTROPYVA:NO`.

See also

EDITBIN Options
/DYNAMICBASE
/LARGEADDRESSAWARE
Windows ISV Software Security Defenses

/INTEGRITYCHECK

Article • 08/03/2021

Specifies that the digital signature of the binary image must be checked at load time.

> **/INTEGRITYCHECK**[**:NO**]

Remarks

In the header of the DLL file or executable file, this option sets a flag that requires a digital signature check by the memory manager to load the image in Windows. Versions of Windows prior to Windows Vista ignore this flag. This option must be set for 64-bit DLLs that implement kernel-mode code, and is recommended for all device drivers. For more information, see Kernel-Mode Code Signing Requirements.

See also

EDITBIN Options

/LARGEADDRESSAWARE

Article • 08/03/2021

```
/LARGEADDRESSAWARE
```

Remarks

This option edits the image to indicate that the application can handle addresses larger than 2 gigabytes.

See also

EDITBIN Options

/NOLOGO (EDITBIN)

Article • 08/03/2021

```
/NOLOGO
```

Remarks

This option suppresses display of the EDITBIN copyright message and version number.

See also

EDITBIN Options

/NXCOMPAT

Article • 08/03/2021

```
/NXCOMPAT[:NO]
```

Remarks

Indicates that an executable was tested to be compatible with the Windows Data Execution Prevention feature.

For more information, see /NXCOMPAT (Compatible with Data Execution Prevention).

See also

EDITBIN Options

/REBASE

Article • 08/03/2021

```
/REBASE[:modifiers]
```

Remarks

This option sets the base addresses for the specified files. EDITBIN assigns new base addresses in a contiguous address space according to the size of each file rounded up to the nearest 64 KB. For details about base addresses, see the Base Address (/BASE) linker option.

Specify the program's executable files and DLLs in the *files* argument on the EDITBIN command line in the order in which they are to be based. You can optionally specify one or more *modifiers*, each separated by a comma (,):

Modifier	Action
BASE=*address*	Provides a beginning address for reassigning base addresses to the files. Specify *address* in decimal or C-language notation. If BASE is not specified, the default starting base address is 0x400000. If DOWN is used, BASE must be specified, and *address* sets the end of the range of base addresses.
BASEFILE	Creates a file named COFFBASE.TXT, which is a text file in the format expected by LINK's /BASE option.
DOWN	Tells EDITBIN to reassign base addresses downward from an ending address. The files are reassigned in the order specified, with the first file located in the highest possible address below the end of the address range. BASE must be used with DOWN to ensure sufficient address space for basing the files. To determine the address space needed by the specified files, run EDITBIN with /REBASE on the files and add 64 KB to the displayed total size.

See also

EDITBIN Options

/RELEASE

Article • 08/03/2021

```
/RELEASE
```

Remarks

This option sets the checksum in the header of an executable file.

The operating system requires the checksum for device drivers. It is recommended that you set the checksum for release versions of your device drivers to ensure compatibility with future operating systems.

See also

EDITBIN Options

/SECTION (EDITBIN)

Article • 08/03/2021

```
/SECTION:name[=newname][,attributes][alignment]
```

Remarks

This option changes the attributes of a section, overriding the attributes that were set when the object file for the section was compiled or linked.

After the colon (**:**), specify the *name* of the section. To change the section name, follow *name* with an equal sign (=) and a *newname* for the section.

To set or change the section's `attributes`, specify a comma (**,**) followed by one or more attributes characters. To negate an attribute, precede its character with an exclamation point (!). The following characters specify memory attributes:

Attribute	**Setting**
c	code
d	discardable
e	executable
i	initialized data
k	cached virtual memory
m	link remove
o	link info
p	paged virtual memory
r	read
s	shared
u	uninitialized data
w	write

To control *alignment*, specify the character **A** followed by one of the following characters to set the size of alignment in bytes, as follows:

Character	Alignment size in bytes
1	1
2	2
4	4
8	8
p	16
t	32
s	64
x	no alignment

Specify the `attributes` and *alignment* characters as a string with no white space. The characters are not case sensitive.

See also

EDITBIN Options

/STACK

Article • 08/03/2021

```
/STACK:reserve[,commit]
```

Remarks

This option sets the size of the stack in bytes and takes arguments in decimal or C-language notation. The /STACK option applies only to an executable file.

The *reserve* argument specifies the total stack allocation in virtual memory. EDITBIN rounds up the specified value to the nearest 4 bytes.

The optional `commit` argument is subject to interpretation by the operating system. In Windows NT, Windows 95, and Windows 98, `commit` specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A higher `commit` value saves time when the application needs more stack space but increases the memory requirements and possibly startup time.

See also

EDITBIN Options

/SUBSYSTEM

Article • 08/03/2021

Specifies the execution environment that's required by the executable image.

```
/SUBSYSTEM:{BOOT_APPLICATION|CONSOLE|EFI_APPLICATION|
        EFI_BOOT_SERVICE_DRIVER|EFI_ROM|EFI_RUNTIME_DRIVER|
        NATIVE|POSIX|WINDOWS|WINDOWSCE}[,major[.minor]]
```

Remarks

This option edits the image to indicate which subsystem the operating system must invoke for execution.

You can specify any of the following subsystems:

BOOT_APPLICATION
An application that runs in the Windows boot environment. For more information about boot applications, see About the BCD WMI Provider.

CONSOLE
A Windows character-mode application. The operating system provides a console for console applications.

EFI_APPLICATION
EFI_BOOT_SERVICE_DRIVER
EFI_ROM
EFI_RUNTIME_DRIVER
Extensible Firmware Interface (EFI) Image

The EFI subsystem options describe executable images that run in the Extensible Firmware Interface environment. This environment is typically provided with the hardware and executes before the operating system is loaded. The major differences between EFI image types are the memory location that the image is loaded into and the action that's taken when the call to the image returns. An EFI_APPLICATION image is unloaded when control returns. An EFI_BOOT_SERVICE_DRIVER or EFI_RUNTIME_DRIVER is unloaded only if control returns with an error code. An EFI_ROM image is executed from ROM. For more information, see the specifications on the Unified EFI Forum website.

NATIVE
Code that runs without a subsystem environment—for example, kernel mode device drivers and native system processes. This option is usually reserved for Windows system features.

POSIX
An app that runs in the POSIX subsystem in Windows.

WINDOWS
An app that runs in the Windows graphical environment. This includes both desktop apps and Universal Windows Platform (UWP) apps.

WINDOWSCE
The WINDOWSCE subsystem indicates that the app is intended to run on a device that has a version of the Windows CE kernel. Versions of the kernel include PocketPC, Windows Mobile, Windows Phone 7, Windows CE V1.0-6.0R3, and Windows Embedded Compact 7.

The optional `major` and `minor` values specify the minimum required version of the specified subsystem:

- The whole number part of the version number—the portion to the left of the decimal point—is represented by `major`.
- The fractional part of the version number—the portion to the right of the decimal point—is represented by `minor`.
- The values of `major` and `minor` must be from 0 through 65,535.

The choice of subsystem affects the default starting address for the program. For more information, see /ENTRY (Entry-Point Symbol), the linker /ENTRY:*function* option.

For more information, including the minimum and default values for the major and minor version numbers for each subsystem, see the /SUBSYSTEM linker option.

See also

EDITBIN Options

/SWAPRUN

Article • 08/03/2021

```
/SWAPRUN:{[!]NET|[!]CD}
```

Remarks

This option edits the image to tell the operating system to copy the image to a swap file and run it from there. Use this option for images that reside on networks or removable media.

You can add or remove the NET or CD qualifiers:

- NET indicates that the image resides on a network.
- CD indicates that the image resides on a CD-ROM or similar removable medium.
- Use !NET and !CD to reverse the effects of NET and CD.

See also

EDITBIN Options

/TSAWARE

Article • 08/03/2021

```
/TSAWARE[:NO]
```

Remarks

The /TSAWARE option to the EDITBIN utility allows you to modify a program image the same way as if you had used the /TSAWARE linker option.

See also

EDITBIN Options

/VERSION

Article • 08/03/2021

```
/VERSION:left[,right]
```

Remarks

This option places a version number into the header of the image.

The whole number part of the version number, the portion to the left of the decimal point, is represented by `left`. The fractional part of the version number, the portion to the right of the decimal point, is represented by `right`.

See also

EDITBIN Options

DUMPBIN Reference

Article • 08/10/2021

The Microsoft COFF Binary File Dumper (DUMPBIN.EXE) displays information about Common Object File Format (COFF) binary files. You can use DUMPBIN to examine COFF object files, standard libraries of COFF objects, executable files, and dynamic-link libraries (DLLs).

> **Note**
>
> We recommend you run DUMPBIN from the Visual Studio command prompt. You can't start it from a system command prompt unless you set the environment correctly. For more information, see **Use the Microsoft C++ toolset from the command line**.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

- DUMPBIN command line
- DUMPBIN options

See also

Additional MSVC Build Tools

DUMPBIN Command Line

Article • 08/03/2021

To run DUMPBIN, use the following syntax:

```
DUMPBIN [options] files...
```

Specify one or more binary files, along with any options required to control the information. DUMPBIN displays the information to standard output. You can either redirect it to a file or use the /OUT option to specify a file name for the output.

When you run DUMPBIN on a file without specifying an option, DUMPBIN displays the /SUMMARY output.

When you type the command `dumpbin` without any other command-line input, DUMPBIN displays a usage statement that summarizes its options.

See also

Additional MSVC Build Tools
DUMPBIN Reference

DUMPBIN options

Article • 08/03/2021

An option consists of an *option specifier*, which is either a dash (`-`) or a forward slash (`/`), followed by the name of the option. Option names can't be abbreviated. Some options take arguments, specified after a colon (`:`). No spaces or tabs are allowed within an option specification. Use one or more spaces or tabs to separate option specifications on the command line. Option names and their keyword or file name arguments aren't case-sensitive. Most options apply to all binary files, but a few apply only to certain types of files. By default, DUMPBIN sends information to standard output. Use the /OUT option to send output to a file.

Options list

DUMPBIN has the following options:

- /ALL
- /ARCHIVEMEMBERS
- /CLRHEADER
- /DEPENDENTS
- /DIRECTIVES
- /DISASM[:{BYTES|NOBYTES}]
- /ERRORREPORT:{NONE|PROMPT|QUEUE|SEND} (Deprecated)
- /EXPORTS
- /FPO
- /HEADERS
- /IMPORTS[:filename]
- /LINENUMBERS
- /LINKERMEMBER[:{1|2}]
- /LOADCONFIG
- /NOPDB

- /OUT:filename
- /PDATA
- /PDBPATH[:VERBOSE]
- /RANGEE:vaMin[,vaMax]
- /RAWDATA[:{NONE|1|2|4|8}[,#]]
- /RELOCATIONS
- /SECTION:name
- /SUMMARY
- /SYMBOLS
- /TLS

To list the options supported by DUMPBIN on the command line, use the **/?** option.

See also

Additional MSVC build tools
DUMPBIN command line
DUMPBIN reference

/ALL

Article • 08/03/2021

```
/ALL
```

Remarks

This option displays all available information except code disassembly. Use /DISASM to display disassembly. You can use /RAWDATA:NONE with /ALL to omit the raw binary details of the file.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/ARCHIVEMEMBERS

Article • 08/03/2021

```
/ARCHIVEMEMBERS
```

Remarks

This option displays minimal information about member objects in a library.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/CLRHEADER

Article • 08/03/2021

Display CLR-specific information.

Syntax

> /CLRHEADER *file*

Arguments

file
An image file built with /clr.

Remarks

/CLRHEADER displays information about the .NET headers used in any managed program. The output shows the location and size, in bytes, of the .NET header and sections in the header.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

When **/CLRHEADER** is used on a file that was compiled with /clr, there will be a **clr Header:** section in the dumpbin output. The value of **flags** indicates which /clr option was used:

- 0 -- /clr (image may contain native code).

You can also programmatically check if an image was built for the common language runtime. For more information, see How to: Determine if an Image is Native or CLR.

The **/clr:pure** and **/clr:safe** compiler options are deprecated in Visual Studio 2015 and unsupported in Visual Studio 2017 and later. Code that must be "pure" or "safe" should be ported to C#.

See also

- DUMPBIN Options

/DEPENDENTS

Article • 08/03/2021

Dumps the names of the DLLs from which the image imports functions. You can use the list to determine which DLLs to redistribute with your app, or find the name of a missing dependency.

Syntax

> **/DEPENDENTS**

This option applies to all the executable files specified on the command line. It doesn't take any arguments.

Remarks

The **/DEPENDENTS** option adds the names of the DLLs from which the image imports functions to the output. This option does not dump the names of the imported functions. To see the names of the imported functions, use the /IMPORTS option.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

Example

This example shows the DUMPBIN output of the **/DEPENDENTS** option on the client executable built in Walkthrough: Create and use your own Dynamic Link Library:

Windows Command Prompt

```
C:\Users\username\Source\Repos\MathClient\Debug>dumpbin /DEPENDENTS
MathClient.exe
Microsoft (R) COFF/PE Dumper Version 14.16.27032.1
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file MathClient1322.exe

File Type: EXECUTABLE IMAGE

  Image has the following dependencies:
```

```
    MathLibrary.dll
    MSVCP140D.dll
    VCRUNTIME140D.dll
    ucrtbased.dll
    KERNEL32.dll

  Summary

        1000 .00cfg
        1000 .data
        2000 .idata
        1000 .msvcjmc
        3000 .rdata
        1000 .reloc
        1000 .rsrc
        8000 .text
       10000 .textbss
```

See also

DUMPBIN Options

/DIRECTIVES

Article • 01/05/2022

```
/DIRECTIVES
```

Remarks

This option dumps the compiler-generated .drectve section of an image.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/DISASM

Article • 08/03/2021

Print the disassembly of code sections in the DUMPBIN output.

Syntax

> **/DISASM**{:[**BYTES**|**NOBYTES**]}

Arguments

BYTES
Includes the instruction bytes together with the interpreted opcodes and arguments in the disassembly output. This is the default option.

NOBYTES
Does not include the instruction bytes in the disassembly output.

Remarks

The **/DISASM** option displays disassembly of code sections in the file. It uses debug symbols if they are present in the file.

/DISASM should only be used on native, not managed, images. The equivalent tool for managed code is ILDASM.

Only the /HEADERS DUMPBIN option is available for use on files produced by the /GL (Whole program optimization) compiler option.

See also

DUMPBIN Options

/ERRORREPORT (dumpbin.exe)

Article • 08/03/2021

> **Note**
>
> The /ERRORREPORT option is deprecated. Starting in Windows Vista, error reporting is controlled by **Windows Error Reporting (WER)** settings.

Syntax

> **/ERRORREPORT**[**NONE** | **PROMPT** | **QUEUE** | **SEND**]

Remarks

The **/ERRORREPORT** arguments are overridden by the Windows Error Reporting service settings. DUMPBIN automatically sends reports of internal errors to Microsoft, if reporting is enabled by Windows Error Reporting. No report is sent if disabled by Windows Error Reporting.

See also

DUMPBIN Options

/EXPORTS

Article • 08/03/2021

```
/EXPORTS
```

Remarks

This option displays all definitions exported from an executable file or DLL.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/FPO

Article • 08/03/2021

```
/FPO
```

Remarks

This option displays frame pointer optimization (FPO) records.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/HEADERS

Article • 08/03/2021

```
/HEADERS
```

Remarks

This option displays the file header and the header for each section. When used with a library, it displays the header for each member object.

Only the **/HEADERS** DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/IMPORTS (DUMPBIN)

Article • 08/03/2021

```
/IMPORTS[:file]
```

This option displays the list of DLLs (both statically linked and delay loaded) that are imported to an executable file or DLL and all the individual imports from each of these DLLs.

The optional `file` specification allows you to specify that the imports for only that DLL will be displayed. For example:

```
dumpbin /IMPORTS:msvcrt.dll
```

Remarks

The output displayed by this option is similar to the /EXPORTS output.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/LINENUMBERS

Article • 08/03/2021

```
/LINENUMBERS
```

Remarks

This option displays COFF line numbers. Line numbers exist in an object file if it was compiled with Program Database (/Zi), C7 Compatible (/Z7), or Line Numbers Only (/Zd). An executable file or DLL contains COFF line numbers if it was linked with Generate Debug Info (/DEBUG).

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/LINKERMEMBER

Article • 08/03/2021

```
/LINKERMEMBER[:{1|2}]
```

Remarks

This option displays public symbols defined in a library. Specify the 1 argument to display symbols in object order, along with their offsets. Specify the 2 argument to display offsets and index numbers of objects, and then list the symbols in alphabetical order, along with the object index for each. To get both outputs, specify /LINKERMEMBER without the number argument.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/LOADCONFIG

Article • 08/03/2021

```
/LOADCONFIG
```

Remarks

This option dumps the IMAGE_LOAD_CONFIG_DIRECTORY structure, an optional structure that is used by the Windows NT loader and defined in WINNT.H.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/NOPDB

Article • 08/03/2021

Tells DUMPBIN not to load and search program database (PDB) files for symbol information.

Syntax

> **/NOPDB**

Remarks

By default, DUMPBIN attempts to load PDB files for its target executables. DUMPBIN uses this information to match addresses to symbol names. The process can be time-consuming if the PDB files are large, or must be loaded from a remote server. The **/NOPDB** option tells DUMPBIN to skip this step. It only prints the addresses and symbol information available in the executable.

To set the /NOPDB linker option in Visual Studio

1. Open the **Property Pages** dialog box for the project. For more information, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **Linker** > **Command Line** property page.
3. In the **Additional options** box, add the **/NOPDB** option. Choose **OK** or **Apply** to save your changes.

To set this linker option programmatically

- This option doesn't have a programmatic equivalent.

See also

DUMPBIN command line
DUMPBIN options
DUMPBIN reference

/OUT (DUMPBIN)

Article • 08/03/2021

```
/OUT:filename
```

Remarks

This option specifies a *filename* for the output. By default, DUMPBIN displays the information to standard output.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/PDATA

Article • 08/03/2021

```
/PDATA
```

Remarks

RISC processors only.

This option dumps the exception tables (.pdata) from an image or object.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/PDBPATH

Article • 03/12/2024

```
/PDBPATH[:VERBOSE] filename
```

Parameters

filename
The name of the .dll or `.exe` file for which you want to find the matching `.pdb` file.

`:VERBOSE`
(Optional) Reports all directories where an attempt was made to locate the `.pdb` file.

Remarks

`/PDBPATH` searches your computer along the same paths that the debugger searches for a `.pdb` file and reports which, if any, `.pdb` files correspond to the file specified in *filename*.

When using the Visual Studio debugger, you may experience a problem because the debugger is using a `.pdb` file for a different version of the file you're debugging.

`/PDBPATH` will search for `.pdb` files along the following paths:

- Check the location where the executable resides.
- Check the location of the PDB written into the executable. This is usually the location at the time the image was linked.
- Check along the search path configured in the Visual Studio IDE.
- Check along the paths in the `_NT_SYMBOL_PATH` and `_NT_ALT_SYMBOL_PATH` environment variables.
- Check in the Windows directory.

See also

DUMPBIN Options
/PDBALTPATH (Use Alternate PDB Path)

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

/RANGE

Article • 08/03/2021

Modifies the output of dumpbin when used with other dumpbin options, such as /RAWDATA or /DISASM.

Syntax

```
/RANGE:vaMin[,vaMax]
```

Parameters

vaMin
The virtual address at which you want the dumpbin operation to begin.

vaMax
(Optional) The virtual address at which you want the dumpbin operation to end. If not specified, dumpbin will go to the end of the file.

Remarks

To see the virtual addresses for an image, use the map file for the image (RVA + Base), the **/DISASM** or **/HEADERS** option of dumpbin, or the disassembly window in the Visual Studio debugger.

Example

In this example, **/range** is used to modify the display of the **/disasm** option. In this example, the starting value is expressed as a decimal number and the ending value is specified as a hex number.

```
dumpbin /disasm /range:4219334,0x004061CD t.exe
```

See also

DUMPBIN Options

/RAWDATA

Article • 08/03/2021

```
/RAWDATA[:{1|2|4|8|NONE[,number]]
```

Remarks

This option displays the raw contents of each section in the file. The arguments control the format of the display, as shown below:

Argument	Result
1	The default. Contents are displayed in hexadecimal bytes, and also as ASCII characters if they have a printed representation.
2	Contents are displayed as hexadecimal 2-byte values.
4	Contents are displayed as hexadecimal 4-byte values.
8	Contents are displayed as hexadecimal 8-byte values.
NONE	Raw data is suppressed. This argument is useful to control the output of /ALL.
Number	Displayed lines are set to a width that holds `number` values per line.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/RELOCATIONS

Article • 08/03/2021

```
/RELOCATIONS
```

Remarks

This option displays any relocations in the object or image.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/SECTION (DUMPBIN)

Article • 08/03/2021

```
/SECTION:section
```

Remarks

This option restricts the output to information on the specified *section*. Use the /HEADERS option to get a list of sections in the file.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/SUMMARY

Article • 08/03/2021

```
/SUMMARY
```

Remarks

This option displays minimal information about sections, including total size. This option is the default if no other option is specified.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/SYMBOLS

Article • 08/03/2021

```
/SYMBOLS
```

This option displays the COFF symbol table. Symbol tables exist in all object files. A COFF symbol table appears in an image file only if it is linked with /DEBUG.

The following is a description of the output for /SYMBOLS. Additional information on the meaning of /SYMBOLS output can be found by looking in winnt.h (IMAGE_SYMBOL and IMAGE_AUX_SYMBOL), or COFF documentation.

Given the following sample dump:

```
Dump of file main.obj
File Type: COFF OBJECT

COFF SYMBOL TABLE
000 00000000 DEBUG    notype       Filename     | .file
    main.cpp
002 000B1FDB ABS      notype       Static       | @comp.id
003 00000000 SECT1    notype       Static       | .drectve
    Section length     26, #relocs    0, #linenums    0, checksum 722C964F
005 00000000 SECT2    notype       Static       | .text
    Section length     23, #relocs    1, #linenums    0, checksum 459FF65F,
selection    1 (pick no duplicates)
007 00000000 SECT2    notype ()    External     | _main
008 00000000 UNDEF    notype ()    External     | ?MyDump@@YAXXZ (void
__cdecl MyDump(void))

String Table Size = 0x10 bytes

  Summary

         26 .drectve
         23 .text
```

Remarks

The following description, for lines that begin with a symbol number, describes columns that have information relevant to users:

- The first three-digit number is the symbol index/number.
- If the third column contains SECT*x*, the symbol is defined in that section of the object file. But if UNDEF appears, it is not defined in that object and must be resolved elsewhere.
- The fifth column (Static, External) tells whether the symbol is visible only within that object, or whether it is public (visible externally). A Static symbol, _sym, wouldn't be linked to a Public symbol _sym; these would be two different instances of functions named _sym.

The last column in a numbered line is the symbol name, both decorated and undecorated.

Only the /HEADERS DUMPBIN option is available for use on files produced with the /GL compiler option.

See also

DUMPBIN Options

/TLS

Article • 08/03/2021

Displays the IMAGE_TLS_DIRECTORY structure from an executable.

Remarks

/TLS displays the fields of the TLS structure as well as the addresses of the TLS callback functions.

If a program does not use thread local storage, its image will not contain a TLS structure. See thread for more information.

IMAGE_TLS_DIRECTORY is defined in winnt.h.

See also

DUMPBIN Options

ERRLOOK Reference

Article • 08/03/2021

The ERRLOOK utility, which is available from the Tools menu as **Error Lookup**, retrieves a system error message or module error message based on the value entered. ERRLOOK retrieves the error message text automatically if you drag and drop a hexadecimal or decimal value from the Visual Studio debugger into the **Value** edit control. You can also enter a value either by typing it in the **Value** edit control or by pasting it from the Clipboard and clicking **Look Up**.

The accelerator keys for Copy (CTRL+C), Cut (CTRL+X), and Paste (CTRL+V) work for both the **Value** and **Error Message** edit controls if you first highlight the text.

In This Section

Value Edit Control
Describes the Value Edit control in ERRLOOK.

Error Message Edit Control
Describes the Error Message Edit control in ERRLOOK.

Modules Button
Describes the Modules button in ERRLOOK.

Look Up Button
Describes the Look Up button in ERRLOOK.

Related Sections

Additional MSVC Build Tools
Provides links to topics discussing the C/C++ build tools provided in Visual C++.

Value Edit Control

Article • 08/03/2021

To use the control, enter a value, paste it from the Clipboard, or drag and drop it from the debugger into this edit control. Enter the value in hexadecimal or decimal form and then click **Look Up**. Hexadecimal values should be preceded by 0x; valid characters are 0-9, A-F, and a-f. Decimal values can be preceded by the minus sign (-); valid characters are 0-9.

See also

ERRLOOK Reference

Error Message Edit Control

Article • 08/03/2021

The **Error Message** box contains the text of the system error message or module error message based on the value entered.

See also

Value Edit Control

Modules Button

Article • 08/03/2021

Click the **Modules** button to bring up the **Additional Modules for Error Searching** dialog. Enter the name of the desired EXE or DLL in the edit box and click **Add** to include the modules in your error message search. Remove a module from the list by highlighting it and clicking the **Remove** button.

See also

Value Edit Control

Look Up Button

Article • 08/03/2021

Click **Look Up** to retrieve the error message that corresponds to the system or module value entered. Values can be entered in hexadecimal or decimal form (including negative decimal values). Modules listed in the **Additional Modules for Error Searching** dialog are also searched.

See also

Value Edit Control

XDCMake Reference

Article • 12/03/2021

The xdcmake.exe program compiles XDC files into an XML file. An XDC file is created by the MSVC compiler for each source code file when source code is compiled with /doc and when the source code file contains documentation comments marked up with XML tags.

To use xdcmake.exe in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Set C++ compiler and build properties in Visual Studio.
2. Select the **Configuration Properties** > **XML Document Comments** property page.
3. Enter options in the property page.

> **Note**
>
> xdcmake.exe options at the command line differ from the options when xdcmake.exe is used in the development environment (property pages). For information on using xdcmake.exe in the development environment, see **XML Document Generator Tool Property Pages**.

Syntax

```
xdcmake input_filename options
```

Parameters

input_filename

The file name of the XDC files used as input to xdcmake.exe. Specify one or more XDC files or use `*.xdc` to use all XDC files in the current directory.

options

You can supply zero or more of the following options:

Option	Description
`/?`, `/help`	Display help for xdcmake.exe.
`/assembly:filename`	Lets you specify the value of the `<assembly>` tag in the XML file. By default, the value of the `<assembly>` tag is the same as the filename of the XML file.
`/nologo`	Suppress copyright message.
`/out:filename`	Lets you specify the name of the XML file. By default, the name of the XML file is the filename of the first XDC file processed by xdcmake.exe.

Remarks

Visual Studio invokes xdcmake.exe automatically when building a project. You can also invoke xdcmake.exe at the command line.

For more information on adding documentation comments to source code files, see Recommended tags for documentation comments.

See also

XML documentation

BSCMAKE Reference

Article • 08/03/2021

> ⚠ **Warning**
>
> Although BSCMAKE is still installed with Visual Studio, it is no longer used by the IDE. Since Visual Studio 2008, browse and symbol information is stored automatically in a SQL Server .sdf file in the solution folder.

The Microsoft Browse Information Maintenance Utility (BSCMAKE.EXE) builds a browse information file (.bsc) from .sbr files created during compilation. Certain third-party tools use .bsc files for code analysis.

When you build your program, you can create a browse information file for your program automatically, using BSCMAKE to build the file. You do not need to know how to run BSCMAKE if you create your browse information file in the Visual Studio development environment. However, you may want to read this topic to understand the choices available.

If you build your program outside of the development environment, you can still create a custom .bsc that you can examine in the environment. Run BSCMAKE on the .sbr files that you created during compilation.

> ⓘ **Note**
>
> You can start this tool only from the Visual Studio Developer command prompt. You cannot start it from a system command prompt or from File Explorer.

This section includes the following topics:

- Building Browse Information Files: Overview
- Building a .bsc file
- BSCMAKE command line
- BSCMAKE command file
- BSCMAKE options
- BSCMAKE exit codes

See also

Additional MSVC Build Tools

Building Browse Information Files: Overview

Article • 03/02/2022

> **⚠ Warning**
>
> Although BSCMAKE is still installed with Visual Studio, it's no longer used by the IDE. Since Visual Studio 2008, browse and symbol information is stored automatically in a SQL Server `.sdf` file in the solution folder.

To create browse information for symbol browsing, the compiler creates an `.sbr` file for each source file in your project, then BSCMAKE.EXE concatenates the `.sbr` files into one `.bsc` file.

Generating `.sbr` and `.bsc` files takes time, so Visual Studio turns off these functions by default. If you want to browse current information, you must turn on the browse options and build your project again.

Use /FR or /Fr to tell the compiler to create `.sbr` files. To create `.bsc` files, you can call BSCMAKE from the command line. Using BSCMAKE from the command line gives you more precise control over the manipulation of browse information files. For more information, see BSCMAKE reference.

> **Tip**
>
> You can turn on `.sbr` file generation but leave `.bsc` file generation turned off. This provides fast builds but also enables you to create a fresh `.bsc` file quickly by turning on `.bsc` file generation and building the project.

You can reduce the time, memory, and disk space required to build a `.bsc` file by reducing the size of the `.bsc` file.

See General Property Page (Project) for information on how to build a browser file in the development environment.

To create a smaller `.bsc` file

1. Use BSCMAKE command-line options to exclude information from the browse information file.

2. Omit local symbols in one or more `.sbr` files when compiling or assembling.

3. If an object file doesn't contain information needed for your current stage of debugging, omit its `.sbr` file from the BSCMAKE command when you rebuild the browse information file.

To combine the browse information from several projects into one browser file (`.bsc`)

1. Either don't build the `.bsc` file at the project level or use the `/n` switch to prevent the `.sbr` files from being truncated.

2. After all the projects are built, run BSCMAKE with all of the `.sbr` files as input. Wildcards are accepted. For instance, if you had project directories `C:\X`, `C:\Y`, and `C:\Z` with `.sbr` files in them and you wanted to combine them all into one `.bsc` file, then use `BSCMAKE C:\X\*.sbr C:\Y\*.sbr C:\Z\*.sbr /o c:\whatever_directory\combined.bsc` to build the combined `.bsc` file.

See also

Additional MSVC build tools
BSCMAKE reference

Building a .Bsc File

Article • 08/03/2021

BSCMAKE can build a new browse information file from newly created .sbr files. It can also maintain an existing .bsc file using .sbr files for object files that have changed since the last build.

- How to create an .sbr file
- How BSCMAKE builds a .bsc file

See also

BSCMAKE Reference

Creating an .Sbr File

Article • 08/03/2021

> ⚠ **Warning**
>
> Although BSCMAKE is still installed with Visual Studio, it is no longer used by the IDE. Since Visual Studio 2008, browse and symbol information is stored automatically in a SQL Server .sdf file in the solution folder.

The input files for BSCMAKE are .sbr files. The compiler creates an .sbr file for each object file (.obj) it compiles. When you build or update your browse information file, all .sbr files for your project must be available on disk.

To create an .sbr file with all possible information, specify /FR.

To create an .sbr file that doesn't contain local symbols, specify /Fr. If the .sbr files contain local symbols, you can still omit them from the .bsc file by using BSCMAKE's /El option.

You can create an .sbr file without performing a full compile. For example, you can specify the /Zs option to the compiler to perform a syntax check and still generate an .sbr file if you specify /FR or /Fr.

The build process can be more efficient if the .sbr files are first packed to remove unreferenced definitions. The compiler automatically packs .sbr files.

See also

Building a .Bsc File

How BSCMAKE Builds a .Bsc File

Article • 08/03/2021

BSCMAKE builds or rebuilds a .bsc file in the most efficient way it can. To avoid potential problems, it is important to understand the build process.

When BSCMAKE builds a browse information file, it truncates the .sbr files to zero length. During a subsequent build of the same file, a zero-length (or empty) .sbr file tells BSCMAKE that the .sbr file has no new contribution to make. It lets BSCMAKE know that an update of that part of the file is not required and an incremental build will be sufficient. During every build (unless the /n option is specified), BSCMAKE first attempts to update the file incrementally by using only those .sbr files that have changed.

BSCMAKE looks for a .bsc file that has the name specified with the /o option. If /o is not specified, BSCMAKE looks for a file that has the base name of the first .sbr file and a .bsc extension. If the file exists, BSCMAKE performs an incremental build of the browse information file using only the contributing .sbr files. If the file does not exist, BSCMAKE performs a full build using all .sbr files. The rules for builds are as follows:

- For a full build to succeed, all specified .sbr files must exist and must not be truncated. If an .sbr file is truncated, you must rebuild it (by recompiling or assembling) before running BSCMAKE.
- For an incremental build to succeed, the .bsc file must exist. All contributing .sbr files, even empty files, must exist and must be specified on the BSCMAKE command line. If you omit an .sbr file from the command line, BSCMAKE removes its contribution from the file.

See also

Building a .Bsc File

BSCMAKE Command Line

Article • 08/03/2021

> **⚠ Warning**
>
> Although BSCMAKE is still installed with Visual Studio, it is no longer used by the IDE. Since Visual Studio 2008, browse and symbol information is stored automatically in a SQL Server .sdf file in the solution folder.

To run BSCMAKE, use the following command line syntax:

```
BSCMAKE [options] sbrfiles
```

Options can appear only in the `options` field on the command line.

The *sbrfiles* field specifies one or more .sbr files created by a compiler or assembler. Separate the names of .sbr files with spaces or tabs. You must specify the extension; there is no default. You can specify a path with the filename, and you can use operating-system wildcards (* and ?).

During an incremental build, you can specify new .sbr files that were not part of the original build. If you want all contributions to remain in the browse information file, you must specify all .sbr files (including truncated files) that were originally used to create the .bsc file. If you omit an .sbr file, that file's contribution to the browse information file is removed.

Do not specify a truncated .sbr file for a full build. A full build requires contributions from all specified .sbr files. Before you perform a full build, recompile the project and create a new .sbr file for each empty file.

The following command runs BSCMAKE to build a file called MAIN.bsc from three .sbr files:

```
BSCMAKE main.sbr file1.sbr file2.sbr
```

For related information, see BSCMAKE Command File and BSCMAKE Options.

See also

BSCMAKE Reference

BSCMAKE Command File (Response File)

Article • 08/03/2021

> **⚠ Warning**
>
> Although BSCMAKE is still installed with Visual Studio, it is no longer used by the IDE. Since Visual Studio 2008, browse and symbol information is stored automatically in a SQL Server .sdf file in the solution folder.

You can provide part or all of the command-line input in a command file. Specify the command file using the following syntax:

```
BSCMAKE @filename
```

Only one command file is allowed. You can specify a path with *filename*. Precede *filename* with an at sign (@). BSCMAKE does not assume an extension. You can specify additional *sbrfiles* on the command line after *filename*. The command file is a text file that contains the input to BSCMAKE in the same order as you would specify it on the command line. Separate the command-line arguments with one or more spaces, tabs, or newline characters.

The following command calls BSCMAKE using a command file:

```
BSCMAKE @prog1.txt
```

The following is a sample command file:

```
/n /v /o main.bsc /El
/S (
toolbox.h
verdate.h c:\src\inc\screen.h
)
file1.sbr file2.sbr file3.sbr file4.sbr
```

See also

BSCMAKE options

Article • 03/22/2022

> ⚠ **Warning**
>
> Although BSCMAKE is still installed with Visual Studio, it's no longer used by the IDE. Since Visual Studio 2008, browse and symbol information is stored automatically in a SQL Server `.sdf` file in the solution folder.

This section describes the options available for controlling BSCMAKE. Several options control the content of the browse information file by excluding or including certain information. The exclusion options can allow BSCMAKE to run faster and may result in a smaller `.bsc` file. Option names are case-sensitive (except for **`/HELP`** and **`/NOLOGO`**).

Only **`/NOLOGO`** and **`/o`** are available from within the Visual Studio development environment. For more information, see Set C++ compiler and build properties in Visual Studio.

Options

`/Ei (` *`filename`* **`...)`**
Excludes the contents of one or more specified *`filename`* include files from the browse information file. To specify multiple files, separate the names with a space and enclose the list in parentheses. Parentheses aren't necessary if you specify only one *`filename`*. Use **`/Ei`** along with the **`/Es`** option to exclude files not excluded by **`/Es`**.

`/El`
Excludes local symbols. The default is to include local symbols. For more information about local symbols, see Creating an .sbr File.

`/Em`
Excludes symbols in the body of macros. Use **`/Em`** to include only the names of macros in the browse information file. The default is to include both the macro names and the result of the macro expansions.

`/Er (` *`symbol`* **`...)`**
Excludes one or more of the specified *`symbol`* symbols from the browse information file. To specify multiple symbol names, separate the names with a space and enclose the list in parentheses. Parentheses are unnecessary if you specify only one *`symbol`*.

`/Es`

Excludes every include file specified with an absolute path, or found in an absolute path specified in the INCLUDE environment variable. (Usually, these files are the system include files, which contain much information you may not need in your browse information file.) This option doesn't exclude files specified without a path, or with relative paths, or files found in a relative path in INCLUDE. You can use the `/Ei` option along with `/Es` to exclude files that `/Es` doesn't exclude. If you want to exclude only some of the files, use `/Ei` instead of `/Es`, and list the files you want to exclude.

`/errorreport:` [`none` | `prompt` | `queue` | `send`]

This option is deprecated. In Windows Vista and later, error reporting is controlled by Windows Error Reporting (WER) settings.

`/HELP`

Displays a summary of the BSCMAKE command-line syntax.

`/Iu`

Includes unreferenced symbols. By default, BSCMAKE doesn't record any symbols that are defined but not referenced. If an `.sbr` file has been packed, this option has no effect for that input file because the compiler has already removed the unreferenced symbols.

`/n`

Forces a non-incremental build. Use `/n` to force a full build of the browse information file whether a `.bsc` file exists or not, and to prevent `.sbr` files from being truncated. For more information, see How BSCMAKE builds a .bsc file.

`/NOLOGO`

Suppresses the BSCMAKE copyright message.

`/o` *`filename`*

The *`filename`* option parameter specifies a name for the browse information file. By default, BSCMAKE gives the browse information file the base name of the first `.sbr` file and a `.bsc` extension.

`/S (` *`filename`* ... `)`

Tells BSCMAKE to process each specified *`filename`* include file the first time it's encountered and to exclude it otherwise. Use this option to save processing time when a file (such as a header, or `.h`, file for a `.c` or `.cpp` source file) is included in several source files but is unchanged by preprocessing directives each time. Use this option if a file is changed in ways unimportant for the browse information file you're creating. To specify multiple files, separate the names with a space, and enclose the list in

parentheses. Parentheses aren't necessary if you specify only one *filename*. If you want to exclude the file every time it's included, use the **/Ei** or **/Es** option.

/v
Provides verbose output, which includes the name of each *.sbr* file being processed and information about the complete BSCMAKE run.

/?
Displays a brief summary of BSCMAKE command-line syntax.

Example

The following command line tells BSCMAKE to do a full build of *main.bsc* from three *.sbr* files. It also tells BSCMAKE to exclude duplicate instances of *toolbox.h*:

Windows Command Prompt

```
BSCMAKE /n /S toolbox.h /o main.bsc file1.sbr file2.sbr file3.sbr
```

See also

BSCMAKE reference

BSCMAKE Exit Codes

Article • 08/03/2021

BSCMAKE returns an exit code (also called a return code or error code) to the operating system or the calling program.

Code	Meaning
0	No error
1	Command-line error
4	Fatal error during build

See also

BSCMAKE Reference

C/C++ Compiler and build tools errors and warnings

Article • 08/03/2021

The articles in this section of the documentation explain diagnostic error and warning messages that are generated by the Microsoft C/C++ compiler and build tools.

> **Important**
>
> The Visual Studio compilers and build tools can report many kinds of errors and warnings. After an error or warning is found, the build tools may make assumptions about code intent and attempt to continue, so that more issues can be reported at the same time. If the tools make the wrong assumption, later errors or warnings may not apply to your project. When you correct issues in your project, always start with the first error or warning that's reported, and rebuild often. One fix may make many subsequent errors go away.

To get help on a particular diagnostic message in Visual Studio, select it in the **Output** window and press the **F1** key. Visual Studio opens the documentation page for that error, if one exists. You can also use the search tool at the top of the page to find articles about specific errors or warnings. Or, browse the list of errors and warnings by tool and type in the table of contents on this page.

> **Note**
>
> Not every Visual Studio error or warning is documented. In many cases, the diagnostic message provides all of the information that's available. If you landed on this page when you used **F1** and you think the error or warning message needs additional explanation, let us know. You can use the feedback buttons on this page to raise a documentation issue on **GitHub**. If you think the error or warning is wrong, or you've found another problem with the toolset, report a product issue on the **Developer Community** site. You can also send feedback and enter bugs within the IDE. In Visual Studio, go to the menu bar and choose **Help** > **Send Feedback** > **Report a Problem**, or submit a suggestion by using **Help** > **Send Feedback** > **Send a Suggestion**.

You may find additional assistance for errors and warnings in Microsoft Learn Q&A forums. Or, search for the error or warning number on the Visual Studio C++ Developer

Community site. You can also search Stack Overflow to find solutions.

For links to additional help and community resources, see Visual C++ Help and Community.

In this section

BSCMAKE errors and warnings (BKxxxx)
Errors and warnings generated by the Microsoft Browse Information Maintenance Utility (BSCMAKE.EXE).

Command-line errors and warnings
Errors and warnings generated by the build tools for command-line options issues.

Compiler fatal errors C999 - C1999
Errors that halt the C++ compiler (CL.EXE).

Compiler errors C2001 - C3999
Errors detected by the C++ compiler (CL.EXE).

Compiler warnings C4000 - C5999
Warnings for issues detected by the C++ compiler (CL.EXE).

Compiler warnings by compiler version
A list of the warnings introduced by each compiler version.

C Runtime errors (Rxxxx)
Errors generated at runtime by the C Runtime Library (CRT).

CVTRES errors and warnings (CVTxxxx)
Errors and warnings generated by the Microsoft Resource File To COFF Object Conversion Utility (CVTRES.EXE).

Expression evaluator errors (CXXxxxx)
Errors generated by the debugger and diagnostics tools.

Linker tools errors and warnings (LNKxxxx)
Errors and warnings generated by the linker and related tools (LINK.EXE, LIB.EXE, DUMPBIN.EXE, EDITBIN.EXE).

Math errors (Mxxxx)
Errors generated by the runtime floating-point math library.

NMAKE errors and warnings (Uxxxx)
Errors and warnings generated by the Microsoft makefile tool (NMAKE.EXE).

Profile-Guided Optimization errors and warnings (PGxxxx)
Errors and warnings generated by the Profile-Guided Optimization (PGO) tools.

Project build errors and warnings (PRJxxxx)
Errors and warnings generated by the native C++ Project build system in Visual Studio.

Resource compiler errors and warnings (RCxxxx, RWxxxx)
Errors and warnings generated by the Resource Compiler (RC.EXE).

Vectorizer and parallelizer messages
Diagnostic messages generated by the vectorizer and parallelizer optimization compiler options.

Related sections

Compiler warnings that are off by default

See also

C/C++ Building Reference
Debugging in Visual Studio

XML documentation (Visual C++)

Article • 12/03/2021

In Visual C++, you can add comments to your source code that are processed to an XML documentation file. This file can then be the input to a process that creates documentation for the classes in your code.

In a Visual C++ code file, XML documentation comments must be located directly before a method or type definition. The comments can be used to populate the IntelliSense QuickInfo data tip in the following scenarios:

1. When the code is compiled as a Windows Runtime component with a WINMD file
2. When the source code is included in the current project
3. In a library whose type declarations and implementations are located in the same header file

> **Note**
>
> In the current release, code comments aren't processed on templates or anything containing a template type (for example, a function taking a parameter as a template). Adding such comments will result in undefined behavior.

For details on creating an XML file with documentation comments, see the following articles.

For information about	See
The compiler options to use	/doc
Tags you can use to provide commonly used functionality in documentation	Recommended tags for documentation comments
The ID strings that the compiler produces to identify the constructs in your code	Processing the XML File
How to delimit documentation tags	Delimiters for Visual C++ documentation tags
Generating an XML file from one or more XDC files.	XDCMake reference
Links to information about XML as it relates to Visual Studio feature areas	XML in Visual Studio

If you need to put XML special characters in the text of a documentation comment, you must use XML entities or a CDATA section.

See also

Component extensions for runtime platforms

Recommended tags for documentation comments

Article • 12/03/2021

The MSVC compiler processes documentation comments in your code to create an XDC file for each compiled source file. Then, xdcmake.exe processes the XDC files to create an XML documentation file. Processing the XML file to create documentation is a detail that needs to be implemented at your site.

Tags are processed on constructs such as types and type members.

Tags must immediately precede types or members.

> **Note**
>
> Documentation comments can't be applied to a namespace or on out of line definitions for properties and events; documentation comments must be on the in-class declarations.

The compiler will process any tag that is valid XML. The following tags provide commonly used functionality in user documentation:

<c>
<code>
<example>
<exception>[1]
<include>[1]
<list>
<para>
<param>[1]
<paramref>[1]
<permission>[1]
<remarks>
<returns>
<see>[1]
<seealso>[1]
<summary>
<value>

1. Compiler verifies syntax.

In the current release, the MSVC compiler doesn't support `<paramref>`, a tag that's supported by other Visual Studio compilers. Visual C++ may support `<paramref>` in a future release.

See also

XML documentation

`<c>` documentation tag

Article • 11/23/2021

The `<c>` tag indicates that text within a description should be marked as code. Use <code> to indicate multiple lines as code.

Syntax

C++

```
/// <c>text</c>
```

Parameters

`text`

The text you want to indicate as code.

Remarks

Compile with /doc to process documentation comments to a file.

Example

C++

```
// xml_c_tag.cpp
// compile with: /doc /LD
// post-build command: xdcmake xml_c_tag.xdc

/// Text for class MyClass.
class MyClass {
public:
   int m_i;
   MyClass() : m_i(0) {}

   /// <summary><c>MyMethod</c> is a method in the <c>MyClass</c> class.
   /// </summary>
   int MyMethod(MyClass * a) {
      return a -> m_i;
   }
};
```

See also

XML documentation

`<code>` documentation tag

Article • 12/03/2021

The `<code>` tag gives you a way to indicate one or more lines as code.

Syntax

```
C++
```

```
/// <code>content</code>

/// <code>
/// content
/// content
/// </code>
```

Parameters

content

The text you want marked as code.

Remarks

Use <c> to indicate a portion of text should be marked as code.

Compile with /doc to process documentation comments to a file.

Example

For an example of how to use the `<code>` tag, see <example>.

See also

XML documentation

`<example>` documentation tag

Article • 12/03/2021

The `<example>` tag lets you specify an example of how to use a method or other library member. Commonly, use of this tag would also involve the `<code>` tag.

Syntax

C++

```
/// <example>description</example>
```

Parameters

`description`
A description of the code sample.

Remarks

Compile with /doc to process documentation comments to a file.

Example

C++

```
// xml_example_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_example_tag.dll

/// Text for class MyClass.
public ref class MyClass {
public:
   /// <summary>
   /// GetZero method
   /// </summary>
   /// <example> This sample shows how to call the GetZero method.
   /// <code>
   /// int main()
   /// {
   ///    return GetZero();
   /// }
   /// </code>
   /// </example>
```

```
    static int GetZero() {
        return 0;
    }
};
```

See also

XML documentation

`<exception>` documentation tag

Article • 11/23/2021

The `<exception>` tag lets you specify which exceptions can be thrown. This tag is applied to a method definition.

Syntax

```
C++
```

```
/// <exception cref="member">description</exception>
```

Parameters

`member`
A reference to an exception that's available from the current compilation environment. Using name lookup rules, the compiler checks that the given exception exists, and translates `member` to the canonical element name in the output XML. The compiler issues a warning if it doesn't find `member`.

Enclose the name in single or double quotation marks.

For more information on how to create a `cref` reference to a generic type, see <see>.

`description`
A description.

Remarks

Compile with /doc to process documentation comments to a file.

The MSVC compiler attempts to resolve `cref` references in one pass through the documentation comments. If using the C++ lookup rules, when a symbol isn't found by the compiler the reference is marked as unresolved. For more information, see <seealso>.

Example

```
C++
```

```
// xml_exception_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_exception_tag.dll
using namespace System;

/// Text for class EClass.
public ref class EClass : public Exception {
   // class definition ...
};

/// <exception cref="System.Exception">Thrown when... .</exception>
public ref class TestClass {
   void Test() {
      try {
      }
      catch(EClass^) {
      }
   }
};
```

See also

XML documentation

`<include>` documentation tag

Article • 12/03/2021

The `<include>` tag lets you refer to comments in another file that describe the types and members in your source code. This tag is an alternative to placing documentation comments directly in your source code file. For example, you can use `<include>` to insert standard "boilerplate" comments that are used throughout your team or company.

Syntax

```
C++
```

```
/// <include file='filename' path='tag-path[@name="ID"' />
```

Parameters

`filename`

The name of the file containing the documentation. The file name can be qualified with a path. Enclose the name in single or double quotation marks. The compiler issues a warning if it doesn't find *`filename`*.

`tag-path`

A valid XPath expression that selects the wanted node-set contained in the file.

`name`

The name specifier in the tag that precedes the comments; *`name`* will have an *`ID`*.

`ID`

The ID for the tag that precedes the comments. Enclose the ID in single or double quotation marks.

Remarks

The `<include>` tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize using `<include>`.

Compile with /doc to process documentation comments to a file.

Example

This example uses multiple files. The first file, which uses `<include>`, contains the following documentation comments:

C++

```
// xml_include_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_include_tag.dll

/// <include file='xml_include_tag.doc'
path='MyDocs/MyMembers[@name="test"]/*' />
public ref class Test {
   void TestMethod() {
   }
};

/// <include file='xml_include_tag.doc'
path='MyDocs/MyMembers[@name="test2"]/*' />
public ref class Test2 {
   void Test() {
   }
};
```

The second file, *`xml_include_tag.doc`*, contains the following documentation comments:

XML

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

Program Output

XML

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>t2</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
</summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
</summary>
        </member>
    </members>
</doc>
```

See also

XML documentation

`<list>` and `<listheader>` documentation tags

Article • 12/03/2021

The `<listheader>` block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for term in the heading.

Syntax

XML

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>description</description>
   </listheader>
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

Parameters

`term`

A term to define, which will be defined in *`description`*.

`description`

Either an item in a bullet or numbered list or the definition of a *`term`*.

Remarks

Each item in the list is specified with an `<item>` block. When creating a definition list, you'll need to specify both *`term`* and *`description`*. However, for a table, bulleted list, or numbered list, you only need to supply an entry for *`description`*.

A list or table can have as many `<item>` blocks as needed.

Compile with /doc to process documentation comments to a file.

Example

C++

```
// xml_list_tag.cpp
// compile with: /doc /LD
// post-build command: xdcmake xml_list_tag.dll
/// <remarks>Here is an example of a bulleted list:
/// <list type="bullet">
/// <item>
/// <description>Item 1.</description>
/// </item>
/// <item>
/// <description>Item 2.</description>
/// </item>
/// </list>
/// </remarks>
class MyClass {};
```

See also

XML documentation

`<para>` documentation tag

Article • 12/03/2021

The `<para>` tag is for use inside a tag, such as <summary>, <remarks>, or <returns>, and lets you add structure to the text.

Syntax

```
C++

/// <para>content</para>
```

Parameters

`content`
The text of the paragraph.

Remarks

Compile with /doc to process documentation comments to a file.

Example

See <summary> for an example of using `<para>`.

See also

XML documentation

`<param>` documentation tag

Article • 12/03/2021

The `<param>` tag should be used in the comment for a method declaration to describe one of the parameters for the method.

Syntax

```
C++

/// <param name='param-name'>description</param>
```

Parameters

`param-name`
The name of a method parameter. Enclose the name in single or double quotation marks. The compiler issues a warning if it doesn't find *`param-name`*.

`description`
A description for the parameter.

Remarks

The text for the `<param>` tag will be displayed in IntelliSense, the Object Browser, and in the Code Comment Web Report.

Compile with /doc to process documentation comments to a file.

Example

```
C++

// xml_param_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_param_tag.dll
/// Text for class MyClass.
public ref class MyClass {
   /// <param name="Int1">Used to indicate status.</param>
   void MyMethod(int Int1) {
   }
};
```

See also

XML documentation

`<paramref>` documentation tag

Article • 12/03/2021

The `<paramref>` tag gives you a way to indicate that a word is a parameter. The XML file can be processed to format this parameter in some distinct way.

Syntax

C++

```
/// <paramref name="ref-name"/>
```

Parameters

`ref-name`

The name of the parameter to refer to. Enclose the name in single or double quotation marks. The compiler issues a warning if it doesn't find *`ref-name`*.

Remarks

Compile with /doc to process documentation comments to a file.

Example

C++

```
// xml_paramref_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_paramref_tag.dll
/// Text for class MyClass.
public ref class MyClass {
   /// <summary>MyMethod is a method in the MyClass class.
   /// The <paramref name="Int1"/> parameter takes a number.
   /// </summary>
   void MyMethod(int Int1) {}
};
```

See also

XML documentation

`<permission>` documentation tag

Article • 12/03/2021

The `<permission>` tag lets you document the access of a member. PermissionSet lets you specify access to a member.

Syntax

```
C++

/// <permission cref="member">description</permission>
```

Parameters

`member`
A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and translates *`member`* to the canonical element name in the output XML. Enclose the name in single or double quotation marks.

The compiler issues a warning if it doesn't find *`member`*.

For information on how to create a `cref` reference to a generic type, see <see>.

`description`
A description of the access to the member.

Remarks

Compile with /doc to process documentation comments to a file.

The MSVC compiler will attempt to resolve `cref` references in one pass through the documentation comments. If the compiler doesn't find a symbol when using the C++ lookup rules, the reference will be marked as unresolved. For more information, see <seealso>.

Example

```
C++
```

```
// xml_permission_tag.cpp
// compile with: /clr /doc /LD
// post-build command: xdcmake xml_permission_tag.dll
using namespace System;
/// Text for class TestClass.
public ref class TestClass {
   /// <permission cref="System::Security::PermissionSet">Everyone can
access this method.</permission>
   void Test() {}
};
```

See also

XML documentation

`<remarks>` documentation tag

Article • 12/03/2021

The `<remarks>` tag is used to add information about a type, supplementing the information specified with <summary>. This information is displayed in the Object Browser and in the Code Comment Web Report.

Syntax

C++

```
/// <remarks>description</remarks>
```

Parameters

description

A description of the member.

Remarks

Compile with /doc to process documentation comments to a file.

Example

C++

```
// xml_remarks_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_remarks_tag.dll

using namespace System;

/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public ref class MyClass {};
```

See also

XML documentation

`<returns>` documentation tag

Article • 12/03/2021

The `<returns>` tag should be used in the comment for a method declaration to describe the return value.

Syntax

C++

```
/// <returns>description</returns>
```

Parameters

`description`
A description of the return value.

Remarks

Compile with /doc to process documentation comments to a file.

Example

C++

```
// xml_returns_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_returns_tag.dll

/// Text for class MyClass.
public ref class MyClass {
public:
   /// <returns>Returns zero.</returns>
   int GetZero() { return 0; }
};
```

See also

XML documentation

`<see>` documentation tag

Article • 12/03/2021

The `<see>` tag lets you specify a link from within text. Use <seealso> to indicate text that you might want to appear in a **See also** section.

Syntax

```
C++

/// <see cref="member"/>
```

Parameters

`member`
A reference to a member or field that is available to be called from the current compilation environment. Enclose the name in single or double quotation marks.

The compiler checks that the given code element exists and resolves *`member`* to the element name in the output XML. The compiler issues a warning if it doesn't find *`member`*.

Remarks

Compile with /doc to process documentation comments to a file.

For an example of using `<see>`, see <summary>.

The MSVC compiler will attempt to resolve `cref` references in one pass through the documentation comments. If the compiler doesn't find a symbol when it's using the C++ lookup rules, it marks the reference as unresolved. For more information, see <seealso>.

Example

The following sample shows how you can make `cref` reference to a generic type so the compiler will resolve the reference.

```
C++
```

```
// xml_see_cref_example.cpp
// compile with: /LD /clr /doc
// the following cref shows how to specify the reference, such that,
// the compiler will resolve the reference
/// <see cref="C{T}">
/// </see>
ref class A {};

// the following cref shows another way to specify the reference,
// such that, the compiler will resolve the reference
/// <see cref="C < T >"/>

// the following cref shows how to hard-code the reference
/// <see cref="T:C`1">
/// </see>
ref class B {};

/// <see cref="A">
/// </see>
/// <typeparam name="T"></typeparam>
generic<class T>
ref class C {};
```

See also

XML documentation

`<seealso>` documentation tag

Article • 12/03/2021

The `<seealso>` tag lets you specify the text that you might want to appear in a See Also section. Use <see> to specify a link from within text.

Syntax

```
C++

/// <seealso cref="member"/>
```

Parameters

`member`
A reference to a member or field that is available to be called from the current compilation environment. Enclose the name in single or double quotation marks.

The compiler checks that the given code element exists and resolves *`member`* to the element name in the output XML. The compiler issues a warning if it doesn't find *`member`*.

For information on how to create a `cref` reference to a generic type, see <see>.

Remarks

Compile with /doc to process documentation comments to a file.

See <summary> for an example of using `<seealso>`.

The MSVC compiler attempts to resolve `cref` references in one pass through the documentation comments. If the compiler doesn't find a symbol when using the C++ lookup rules, it marks the reference as unresolved.

Example

In the following sample, an unresolved symbol is referenced in a `cref`. The XML comment for the `cref` to `A::Test` is well formed: `<seealso cref="M:A.Test" />`. However, the `cref` to `B::Test` becomes `<seealso cref="!:B::Test" />`.

C++

```
// xml_seealso_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_seealso_tag.dll

/// Text for class A.
public ref struct A {
   /// <summary><seealso cref="A::Test"/>
   /// <seealso cref="B::Test"/>
   /// </summary>
   void MyMethod(int Int1) {}

   /// text
   void Test() {}
};

/// Text for class B.
public ref struct B {
   void Test() {}
};
```

See also

XML documentation

`<summary>`

Article • 12/03/2021

The `<summary>` tag should be used to describe a type or a type member. Use <remarks> to add supplemental information to a type description.

Syntax

C++

```
/// <summary>description</summary>
```

Parameters

`description`
A summary of the object.

Remarks

The text for the `<summary>` tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser and in the Code Comment Web Report.

Compile with /doc to process documentation comments to a file.

Example

C++

```
// xml_summary_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_summary_tag.dll

/// Text for class MyClass.
public ref class MyClass {
public:
   /// <summary>MyMethod is a method in the MyClass class.
   /// <para>Here's how you could make a second paragraph in a description.
<see cref="System::Console::WriteLine"/> for information about output
statements.</para>
   /// <seealso cref="MyClass::MyMethod2"/>
   /// </summary>
```

```
    void MyMethod(int Int1) {}

    /// text
    void MyMethod2() {}
};
```

See also

XML documentation

`<value>` documentation tag

Article • 12/03/2021

The `<value>` tag lets you describe a property and property accessor methods. When you add a property with a code wizard in the Visual Studio integrated development environment, it will add a <summary> tag for the new property. You need to manually add a `<value>` tag to describe the value that the property represents.

Syntax

C++

```
/// <value>property-description</value>
```

Parameters

`property-description`
A description for the property.

Remarks

Compile with /doc to process documentation comments to a file.

Example

C++

```
// xml_value_tag.cpp
// compile with: /LD /clr /doc
// post-build command: xdcmake xml_value_tag.dll
using namespace System;
/// Text for class Employee.
public ref class Employee {
private:
   String ^ name;
   /// <value>Name accesses the value of the name data member</value>
public:
   property String ^ Name {
      String ^ get() {
         return name;
      }
      void set(String ^ i) {
```

```
            name = i;
        }
    }
};
```

See also

XML documentation

XML documentation file processing

Article • 12/03/2021

The compiler generates an ID string for each construct in your code that is tagged to generate documentation. For more information, see Recommended tags documentation comments. The ID string uniquely identifies the construct. Programs that process the XML file can use the ID string to identify the corresponding .NET Framework metadata or reflection item to which the documentation applies.

The XML file isn't a hierarchical representation of your code, it's a flat list with a generated ID for each element.

The compiler observes the following rules when it generates the ID strings:

- No white space is placed in the string.
- The first part of the ID string identifies the kind of member being identified, with a single character followed by a colon. The following member types are used:

Character	Description
N	Namespace You can't add documentation comments to a namespace, cref references to a namespace are possible.
T	Type: class, interface, struct, enum, delegate
D	Typedef
F	Field
P	Property (including indexers or other indexed properties)
M	Method (including such special methods as constructors, operators, and so forth)
E	Event
!	Error string The rest of the string provides information about the error. The MSVC compiler generates error information for links that can't be resolved.

- The second part of the string is the fully qualified name of the item, starting at the root of the namespace. The name of the item, its enclosing type or types, and namespace are separated by periods. If the name of the item itself has periods,

they're replaced by the hash-sign ('#'). It's assumed that no item has a hash-sign directly in its name. For example, the fully qualified name of the `String` constructor would be `System.String.#ctor`.

- For properties and methods, if there are arguments to the method, the argument list enclosed in parentheses follows. If there are no arguments, no parentheses are present. The arguments are separated by commas. Each argument is encoded the same way it's encoded in a .NET Framework signature:
 - Base types. Regular types (`ELEMENT_TYPE_CLASS` or `ELEMENT_TYPE_VALUETYPE`) are represented as the fully qualified name of the type.
 - Intrinsic types (for example, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_OBJECT`, `ELEMENT_TYPE_STRING`, `ELEMENT_TYPE_TYPEDBYREF`, and `ELEMENT_TYPE_VOID`) are represented as the fully qualified name of the corresponding full type, for example, **`System.Int32`** or **`System.TypedReference`**.
 - `ELEMENT_TYPE_PTR` is represented as a '`*`' following the modified type.
 - `ELEMENT_TYPE_BYREF` is represented as a '`@`' following the modified type.
 - `ELEMENT_TYPE_PINNED` is represented as a '`^`' following the modified type. The MSVC compiler never generates this element.
 - `ELEMENT_TYPE_CMOD_REQ` is represented as a '`|`' and the fully qualified name of the modifier class, following the modified type. The MSVC compiler never generates this element.
 - `ELEMENT_TYPE_CMOD_OPT` is represented as a '`!`' and the fully qualified name of the modifier class, following the modified type.
 - `ELEMENT_TYPE_SZARRAY` is represented as "`[]`" following the element type of the array.
 - `ELEMENT_TYPE_GENERICARRAY` is represented as "`[?]`" following the element type of the array. The MSVC compiler never generates this element.
 - `ELEMENT_TYPE_ARRAY` is represented as `[`lower bound`:`size`,`lower bound`:`size`]` where the number of commas is the rank - 1, and the lower bound and size of each dimension, if known, are represented in decimal. If a lower bound or size isn't specified, it's omitted. If the lower bound and size are omitted for a particular dimension, the '`:`' is omitted as well. For example, a 2-dimensional array with 1 as the lower bound and unspecified sizes is represented as `[1:,1:]`.

- `ELEMENT_TYPE_FNPTR` is represented as "=FUNC:`type`(*signature*)", where `type` is the return type, and *signature* is the arguments of the method. If there are no arguments, the parentheses are omitted. The MSVC compiler never generates this element.

 The following signature components aren't represented because they're never used for differentiating overloaded methods:

 - Calling convention
 - Return type
 - `ELEMENT_TYPE_SENTINEL`

- For conversion operators only, the return value of the method is encoded as a '`~`' followed by the return type, as previously encoded.

- For generic types, the name of the type will be followed by a back tick and then a number that indicates the number of generic type parameters. For example,

 XML

  ```
  <member name="T:MyClass`2">
  ```

 The example shows a type that's defined as `public class MyClass<T, U>`.

 For methods that take generic types as parameters, the generic type parameters are specified as numbers prefaced with back ticks (for example `0, `1). Each number represents a zero-based array position for the type's generic parameters.

Example

The following examples show how the ID strings for a class and its members would be generated.

C++

```
// xml_id_strings.cpp
// compile with: /clr /doc /LD
///
namespace N {
// "N:N"

   /// <see cref="System" />
   //  <see cref="N:System"/>
   ref class X {
```

```
    // "T:N.X"

    protected:
       ///
       !X(){}
       // "M:N.X.Finalize", destructor's representation in metadata

    public:
       ///
       X() {}
       // "M:N.X.#ctor"

       ///
       static X() {}
       // "M:N.X.#cctor"

       ///
       X(int i) {}
       // "M:N.X.#ctor(System.Int32)"

       ///
       ~X() {}
       // "M:N.X.Dispose", Dispose function representation in metadata

       ///
       System::String^ q;
       // "F:N.X.q"

       ///
       double PI;
       // "F:N.X.PI"

       ///
       int f() { return 1; }
       // "M:N.X.f"

       ///
       int bb(System::String ^ s, int % y, void * z) { return 1; }
       // "M:N.X.bb(System.String,System.Int32@,System.Void*)"

       ///
       int gg(array<short> ^ array1, array< int, 2 >^ IntArray) { return 0; }
       // "M:N.X.gg(System.Int16[], System.Int32[0:,0:])"

       ///
       static X^ operator+(X^ x, X^ xx) { return x; }
      // "M:N.X.op_Addition(N.X,N.X)"

       ///
       property int prop;
       // "M:N.X.prop"

       ///
       property int prop2 {
       // "P:N.X.prop2"
```

```
         ///
         int get() { return 0; }
         // M:N.X.get_prop2

         ///
         void set(int i) {}
         // M:N.X.set_prop2(System.Int32)
      }

      ///
      delegate void D(int i);
      // "T:N.X.D"

      ///
      event D ^ d;
      // "E:N.X.d"

      ///
      ref class Nested {};
      // "T:N.X.Nested"

      ///
      static explicit operator System::Int32 (X x) { return 1; }
      //
"M:N.X.op_Explicit(N.X!System.Runtime.CompilerServices.IsByValue)~System.Int
32"
   };
}
```

See also

XML documentation

Delimiters for Visual C++ documentation tags

Article • 12/03/2021

The use of documentation tags requires *delimiters*, which indicate to the compiler where a documentation comment begins and ends.

You can use the following kinds of delimiters with the XML documentation tags:

Delimiter	Description
`///`	This is the form that's shown in documentation examples and used by the Visual Studio C++ project templates.
`/** */`	These are multiline delimiters.

There are some formatting rules when using the `/** */` delimiters:

- For the line that contains the `/**` delimiter, if the rest of the line is whitespace, the line isn't processed for comments. If the first character is whitespace, that whitespace character is ignored and the rest of the line is processed. Otherwise, the entire text of the line after the `/**` delimiter is processed as part of the comment.
- For the line that contains the `*/` delimiter, if there's only whitespace up to the `*/` delimiter, that line is ignored. Otherwise, the text on the line up to the `*/` delimiter is processed as part of the comment, subject to the pattern-matching rules described in the following bullet.
- For the lines after the one that begins with the `/**` delimiter, the compiler looks for a common pattern at the beginning of each line that consists of optional white space and an asterisk (`*`), followed by more optional whitespace. If the compiler finds a common set of characters at the beginning of each line, it will ignore that pattern for all lines after the `/**` delimiter, up to and possibly including the line that contains the `*/` delimiter.

Examples

- The only part of the following comment that will be processed is the line that begins with `<summary>`. The following two tag formats will produce the same comments:

```
C++
```

```
/**
<summary>text</summary>
*/
/** <summary>text</summary> */
```

- The compiler applies a pattern of " * " to ignore at the beginning of the second and third lines.

```
C++
```

```
/**
 * <summary>
 *  text </summary>*/
```

- The compiler finds no pattern in this comment because there's no asterisk on the second line. All text on the second and third lines, up until the `*/`, will be processed as part of the comment.

```
C++
```

```
/**
 * <summary>
   text </summary>*/
```

- The compiler finds no pattern in this comment for two reasons. First, there's no line that begins with a consistent number of spaces before the asterisk. Second, the fifth line begins with a tab, which doesn't match spaces. All text from the second line until the `*/` will be processed as part of the comment.

```
C++
```

```
/**
  * <summary>
  * text
 *  text2
   *  </summary>
*/
```

See also

XML documentation